10th International Winter School on

Modeling and Verifying Parallel Processes

(MOVEP 2012)

Proceedings

edited by Pierre-Alain Reynier

Preface

MOVEP 2012 is the tenth occurrence in the series of MOVEP summer/winter schools devoted to the wide area of modeling and verifying software and hardware systems. MOVEP (MOdeling and VErifying parallel Processes) was originally a French-speaking school and was initiated by A. Arnold (LaBRI, Bordeaux), J. Beauquier (LRI, Orsay), and O. Roux (IRCCyN, Nantes) in 1994. MOVEP was held in Nantes five times from 1994 to 2002, and adopted English as its working language in 2000. In 2004 the school was organized in Brussels, in 2006 in Bordeaux, in 2008 near Orléans, and in 2010 in Aachen. MOVEP 2012 is organized in Marseille, France, and will be hosted by the CIRM, an internationally recognized conference center in mathematics, located on the campus of Luminy. This year, about 95 people from all over Europe will participate to MOVEP.

The program of MOVEP is composed of six tutorials and five advanced lectures that were proposed by the program committee. These lectures cover various topics including model checking, runtime verification, synthesis, real-time and stochastic systems, Petri nets, games, logic and security. The first part of these proceedings contains short/extended abstracts and references for the tutorials and advanced lectures.

Another important part of MOVEP are the sessions devoted to Ph.D. students. In these sessions students have the opportunity to report on their work in short presentations. In MOVEP 2012 there are 21 such presentations in four sessions. These presentations were selected by the organizers on the basis of submitted extended abstracts which can be found in the second part of these proceedings. We hope that the student sessions will help the participants of the school to get feedback on their ongoing work, to get in touch with other researchers from their area and to initiate new collaborations.

We would like to thank the Program Committee members for their help in compiling the well-balanced program of the school. We also thank the local organization committee and the members of the MOVE research team of the LIF who assisted us in the evaluation of the submitted papers. We would also like to thank the invited speakers and the authors of the papers submitted to the student sessions.

We gratefully acknowledge support from CIRM, Laboratoire d'Informatique Fondamentale de Marseille (LIF), Aix-Marseille Université, CNRS, INRIA, Action AFSEC (Formal Approaches for Communicating Embedded Systems) of the CNRS GDR ASR (Architecture, systems and network), Conseil Général des Bouches du Rhône, and Ville de Marseille.

November 2012
F. Cassez, T. Jéron, C. Löding, N. Markey, P.-A. Reynier, M. Ryan
Steering Committee MOVEP 2012

MOVEP 2012 Committees

Steering Committee

Franck Cassez (NICTA, Sydney, AU)
Thierry Jéron (INRIA, Rennes, F)
Christof Löding (RWTH Aachen, D)
Nicolas Markey (LSV, CNRS & ENS Cachan, F)
Pierre-Alain Reynier (LIF, AMU & CNRS, F)
Mark D. Ryan (Univ. Birmingham, UK)

Program Committee

Krishnendu Chatterjee (IST, A)
Alessandro Cimatti (IRST, I)
Véronique Cortier (LORIA, CNRS & INRIA, F)
Giorgio Delzanno (University of Genova, I)
Dino Distefano (Queen Mary University of London, UK)
Martin Fränzle (University of Oldenburg, D)
Petr Jancar (Technical University, Ostrava, CZ)
Claude Jard (IRISA, ENS Cachan & CNRS & INRIA, F)
Bengt Jonsson (Uppsala University, SE)
Joost-Pieter Katoen (RWTH Aachen, D)
Felix Klaedtke (ETH Zurich, Switzerland)
Kim G. Larsen (Aalborg University, DK)
Rupak Majumdar (MPI-SWS & UCLA, D)
Oded Maler (VERIMAG, CNRS, F)
Markus Müller-Olm (University of Münster, D)
Joel Ouaknine (University of Oxford, UK)
Jean-François Raskin (Université Libre de Bruxelles, B)
Olivier H. Roux (IRCCyN, F)
Stefan Schwoon (LSV, CNRS & ENS Cachan, F)
Jeremy Sproston (University of Torino, I)
Grégoire Sutre (LaBRI, CNRS & University of Bordeaux, F)
Frits Vaandrager (Radboud University Nijmegen, NL)
Luca Vigano (University of Verona, I)

Organizing Committee

Arnaud Labourel
Laurent Braud
Rémi Morin
Nicolas Baudru
Mathieu Caralp
Nadine Comes
Martine Quessada
Sylvie Ros

Contents

Tutorials

Technical Talks

Student Papers

Tutorials

Logic and Verification

Moshe Y. Vardi

Rice University, Houston, USA

Mathematical logic developed as an attempt to provide formal foundations for mathematics. The success of that project can be questioned, as the logical foundations of mathematics proved to be incomplete, possibly inconsistent, and undecidable. Logic, on the other hand, proved to be highly successful in providing formal foundations for reasoning about computing systems, where it is deployed today in industrial tools. This tutorial will focus on one application of logic to verification, which is the temporal analysis of systems.

References

[1] Moshe Y. Vardi. From philosophical to industrial logics. Proc. 3rd Indian Conference on Logic and Its Applications. Lecture Notes in AI 5378, Springer, pp. 89-115, 2009.

[2] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Logics for Concurrency: Structure versus Automata. Springer-Verlag, Lecture Notes in Computer Science 1043, 1996, pp. 238–266.

Probabilistic Systems

Marta Kwiatkowska

Department of Computer Science, University of Oxford, UK

Probabilistic model checking is a formal verication technique for the analysis of systems that exhibit stochastic behaviour. Such behaviour occurs, for example, due to component failure or randomisation, commonly used as a symmetry breaker in distributed coordination and communication protocols. The techniques have been implemented in tools such as PRISM (www.prismmodelchecker.org) and enable a range of quantitative analyses of probabilistic models against specifications such as the worst-case probability of failure within 10 seconds or the minimum expected power consumption over all possible schedulings. This course will give an overview of probabilistic model checking discrete-time Markov chains and Markov decision processes, explaining the underlying theory and model checking algorithms for temporal logics such as PCTL and LTL. The material will be illustrated with several case studies that have been modelled and analysed in PRISM.

Unfoldings: A Partial Order approach to Model Checking

Javier Esparza

Institut für Informatik, Technische Universitat München, Garching, Germany

State space methods are the most popular approach to the automatic verification of concurrent systems. In their basic form, these methods explore the transition system associated to the concurrent system. Loosely speaking, the transition system is a graph having the reachable states of the system as nodes, and an edge from a state s to another state s' whenever the system can make a move from s to s'. In the worst case, state space methods need to explore all nodes and transitions of the transition system.

The main problem of transition systems as a basis for state space methods is the well-known state-explosion problem. Imagine a concurrent system consisting of n sequential subsystems, communicating in some way, and assume further that each of these subsystems can be in one out of m possible states. The global state of the concurrent system is given by the local states of its components, and so the system may have up to m^n reachable states; in fact, this bound is already reached by the rather uninteresting system in whose components run independently of each other, without communicating at all. So very small concurrent systems may generate very large transition systems. As a consequence, naive state space methods may have huge time and space requirements even for very small and simple systems.

The unfolding method is a technique for alleviating the state-explosion problem. It uses results of the theory of true concurrency to replace transition systems by special partially ordered graphs. While these graphs contain full information about the reachable states of the system, their nodes are not reachable states themselves. In particular, the number of nodes of the graph does not grow linearly in the number of reachable states. The goal of the course is to provide a gentle introduction to the basics of the unfolding method, and in particular to introduce an unfolding-based algorithm for model checking concurrent systems against properties specified as formulas of Linear Temporal Logic (LTL).

The course is based on the book [1].

References

[1] Javier Esparza and Keijo Heljanko. Unfoldings. A Partial-Order Approach to Model Checking. EATCS Monographs on Theoretical Computer Science. Springer, 2008. `http://www7.in.tum.de/~esparza/bookunf.html`

Timed automata and their quantitative extensions

Kim G. Larsen

Department of Computer Science, Aalborg University, Denmark

Timed automata is by now a well-established formalism for modeling and analyzing real-time systems, including real-time controllers, communication protocols. Over the years a number of symbolic techniques have been developed for the efficient analysis of timed automata and with implementation in the tool suite UPPAAL (`www.uppaal.{com,org}`). The UPPAAL modeling formalism allows for the efficient analysis of safety and (time-bounded) liveness properties of networks of interacting timed automata extended with discrete variables, structured and user-defined types as well as user-defined functions. The course will give an overview of the modeling formalism of timed automata and the basic symbolic model checking algorithms. More recently the formalism of timed automata has been extended with continuous observer variables allowing for issues related to e.g energy consumption in embedded systems. The course reviews a number of results for the resulting notion of priced timed automata, including recent results on energy-bounded infinite runs in the case when energy may both be consumed as well as harvested. Most recently a stochastic semantics of (priced) timed automata has been put forward enabling the expression of performance properties such as the probability of violating a deadline or the expected energy consumption. A range of so-called highly scalable statistical model-checking techniques have been implemented in UPPAAL, allowing estimation and testing of such performance metrics to be obtained through simulation, where the simulation effort increases with the desired level of confidence. The lecture will contain demonstration of the UPPAAL tool, the new statistical model checking engine as well as several case studies that have been dealt with by the tool.

Games for Verification and Synthesis

Hugo Gimbert

CNRS, LaBRI, Université de Bordeaux, France

Game playing is a powerful metaphor that fits many situations in which interaction between autonomous agents plays a central role. Numerous tasks in computer science and AI such as design, synthesis, verification, testing, query evaluation, planning, etc. can be formulated in game-theoretic terms. Viewing them abstractly as games reveals the underlying algorithmic questions, and helps to clarify the relationships between problem domains.

This talk will illustrate how games can be used in several ways in this context: as algorithmically tractable models of controllable open systems (e.g. games on graphs and stochastic games) as algorithmic tools (e.g. for mu-calculus model-checking) as well as proof tools (e.g. to prove stability under complementation of recognizable set of trees).

Safety, Dependability and Performance Analysis of Extended AADL Models

Alessandro Cimatti
Fondazione Bruno Kessler
Trento, Italy
cimatti@fbk.eu

Thomas Noll
RWTH Aachen University
Aachen, Germany
noll@cs.rwth-aachen.de

Abstract

This MOVEP 2012 tutorial presents a component-based modeling approach to system-software co-engineering of real-time embedded systems, in particular aerospace systems. Our method is centered around the standardized Architecture Analysis and Design Language (AADL) modeling framework. Taking the core features of AADL and its recent Error Model Annex, we have set up a modeling framework that supports a variety of system analysis and verification methods. Its major distinguishing aspects are the possibility to describe hardware and software components and their nominal operation, hybrid (and timing) aspects, as well as probabilistic faults and their propagation and recovery. Moreover, it supports dynamic (i.e., on-the-fly) reconfiguration of components and inter-component connections. The operational semantics gives a precise interpretation of specifications by providing a mapping onto networks of event-data automata. These networks are then subject to different kinds of formal analysis such as model checking, safety and dependability analysis, and performance evaluation. We demonstrate tool support realizing these analyses and report on industrial case studies that have been carried out in the context of aerospace systems. The tool is publicly available from [1].

1 Introduction

Hardware/software (HW/SW) co-design of safety-critical systems such as on-board systems that appear in the aerospace domain is a very complex and highly challenging task. Component-based engineering is an important paradigm that is helpful to master this design complexity while, in addition, allowing for re-usability. The key principle in component-based design is a clear distinction between component behaviour (implementation) and the possible interactions between the individual components (interfacing). Components may be structured in a hierarchical manner akin to an AND-composition in the visual modelling formalism Statecharts [21]. The internal structure of a component implementation is specified by its decomposition into subcomponents, together with their HW/SW bindings and their interaction via connections over ports. Component behaviour is typically described by a textual representation of mode-transition diagrams, a kind of finite-state automata.

As safety-critical systems are subject to hardware and software faults, the adequate modelling of faults, their likelihood of occurrence, and the way in which a system can recover from faults, are essential to a model-based approach for safety-critical systems. Although several formal approaches to component-based design have been recently reported in the literature, error handling and modelling have received scant attention, if at all. Another shortcoming of many proposals —a notable exception is the recent work of [12]— is the lack of connection to a notation that is used and understood by design engineers. In the COMPASS Project [1], we attempt to overcome these shortcomings by enriching a

practical component-based modelling approach with appropriate means for modelling probabilistic fault behaviour.

To warrant acceptance by design engineers in, e.g., aerospace industry and the automotive domain, our approach is based on the Architecture Analysis and Design Language (AADL), a design formalism that is standardised by the Society of Automotive Engineers [31, 32]. The major distinguishing aspects of AADL are the possibility to describe nominal hardware and software operations, hybrid (and timing) aspects, as well as dynamic reconfiguration of components and port connections between components. In order to model probabilistic faults, their propagation and recovery, and degraded modes of operation, we adopt the recent AADL Error Model Annex [33].

Error behaviour is defined by probabilistic finite-state machines, where error delays are determined by continuous random variables, in particular, those that are governed by negative exponential distributions. This strongly resembles the well-studied model of continuous-time Markov chains (CTMCs), with the exception that nondeterminism is also allowed in our setting. The integration of nominal behaviour and error models follows an approach advocated in [11], and basically boils down to a product construction of an event-data automaton and a finite interactive Markov chain (i.e., a CTMC exhibiting nondeterminism) [23].

The remainder of this extended abstract is structured as follows. Section 2 provides a gentle introduction to our AADL dialect incorporating error modelling features by means of a small example. Section 3 presents the COMPASS Toolset that supports the modelling and formal analysis of AADL specifications.

2 SLIM Modelling Language

The System-Level Integrated Modelling (SLIM) language [9] has been designed as an "extended subset" of AADL [31, 32] in order to provide a cohesive and uniform approach to model heterogeneous systems, consisting of software (e.g., processes and threads) and hardware (e.g., processors and buses) components, and their interactions. Furthermore, it has been drafted with the following essential features in mind:

- Modelling both the system's nominal and faulty behaviour. To this aim, SLIM provides primitives to describe software and hardware faults, error propagation (that is, turning fault occurrences into failure events), sporadic (transient) and permanent faults, and degraded modes of operation (by mapping failures from architectural to service level).
- Modelling (partial) observability and the associated observability requirements. These notions are essential to deal with diagnosability and Fault Detection, Isolation and Recovery (FDIR) analyses.
- Specifying timed and hybrid behaviour. In particular, in order to analyse continuous physical systems such as mechanics and hydraulics, the SLIM language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modelling probabilistic aspects, such as random faults, repairs, and stochastic timing.

Here we give a comprehensive presentation of the capabilities of the SLIM language, using a running example. Its formal semantics is described in [8].

A complete SLIM specification consists of three parts, namely a description of the nominal behaviour, a description of the error behaviour and a fault injection specification that describes how the error behaviour influences the nominal behaviour. These three parts are discussed in order below.

```
device Battery
 features
  empty: out event port;
  tryReset: in data port bool default false;
  voltage: out data port real default 6.0;
end Battery;

device implementation Battery.Imp
 subcomponents
  energy: data continuous default 1.0;
 modes
  charged: activation mode while energy' = -0.02 and energy >= 0.2;
  depleted: mode while energy' = -0.03 and energy >= 0.0;
 transitions
  charged -[then voltage := 2.0 * energy + 4.0]-> charged;
  charged -[reset when tryReset]-> charged;
  charged -[empty when energy = 0.2]-> depleted;
  depleted -[then voltage := 2.0 * energy + 4.0]->depleted;
  depleted -[reset when tryReset]-> depleted;
end Battery.Imp;
```

Figure 1: Specification of a battery component.

2.1 Nominal Behaviour

A SLIM model is hierarchically organised into *components*, distinguished into software (processes, threads, data), hardware (processors, memories, devices, buses), and composite components (called *systems*). Components are defined by their *type* (specifying the functional interfaces as seen by the environment) and their *implementation* (representing the internal structure). An example of a component's type and implementation is shown in Figure 1, which represents a simple battery device [7].

The component type describes the ports through which the component communicates. For example, the type interface of Figure 1 features three ports, namely an outgoing event port `empty` which indicates that the battery is about to become discharged, an incoming data port `tryReset` which indicates that the battery device should (try to) reset, and an outgoing data port `voltage` which makes its current voltage level accessible to the environment.

A component implementation defines its subcomponents, their interaction through (event and data) port connections, the (physical) bindings at runtime, the operational behaviour via modes, the transitions between them, which are spontaneous or triggered by events arriving at the ports, and the timing and hybrid behaviour of the component. For example, the implementation of Figure 1 specifies the battery to be in the `charged` mode whenever activated, with an `energy` level of 100%. This level is continuously decreased by 2% (of the initial amount) per time unit (`energy'` denotes the first derivative of `energy`) until a threshold value of 20% is reached, upon which the battery changes to the `depleted` mode. This mode transition triggers the `empty` output event, and the loss rate of energy is increased to 3%. Moreover, the `voltage` value is regularly computed from the `energy` level (ranging between 6.0 and 4.0 [volts]) and made accessible to the environment via the corresponding outgoing data port. In addition, the battery

```
system Power
 features
  alert: out data port bool observable;
end Power;

system implementation Power.Imp
 subcomponents
  batt1: device Battery.Imp in modes (primary);
  batt2: device Battery.Imp in modes (backup);
  mon: device Monitor.Imp;
 connections
  data port batt1.voltage -> mon.voltage in modes (primary);
  data port batt2.voltage -> mon.voltage in modes (backup);
  data port mon.alert -> alert;
  data port mon.alert -> batt1.tryReset in modes (primary);
  data port mon.alert -> batt2.tryReset in modes (backup);
 modes
  primary: initial mode;
  backup: mode;
 transitions
  primary -[batt1.empty]-> backup;
  backup -[batt2.empty]-> primary;
end Power.Imp;
```

Figure 2: The complete power system.

listens to the `tryReset` port to decide when a **`reset`** operation should be performed in reaction to faulty behaviour (see the description of error models below).

In general, the mode transition system —basically a finite-state automaton— describes how the component evolves from mode to mode while performing events. Invariants on the values of data components (such as "`energy >= 0.2`" in mode `charged`) restrict the residence time in a mode. Trajectory equations (such as the one associated with `energy'`) specify how continuous variables evolve while residing in a mode. This is akin to timed and hybrid automata [22]. Here we assume that all invariants are given by Boolean expressions over data subcomponents and constants where each arithmetic subexpression is linear. Moreover we constrain the derivatives occurring in trajectory equations to real constants, i.e., the evolution of continuous variables is described by linear functions.

A mode transition is of the form m `-[`e **`when`** g **`then`** f`]->` m'. It asserts that the component can evolve from mode m to mode m' on the occurrence of event e (the trigger event) provided the guard g, a Boolean expression that may depend on the component's (discrete and continuous) data elements, holds. Here "data elements" refers to both (incoming and outgoing) data ports and data subcomponents of the respective component. On transiting, the effect f which may update data subcomponents or outgoing data ports (like `voltage`) is applied. The presence of event e, guard **`when`** g and effect **`then`** f is optional. If absent, e defaults to an internal event, g to **`true`**, and f to the empty effect.

Mode transitions may give rise to modifications of a component's configuration: subcomponents can become (de-)activated and port connections can be (de-)established. This depends on the **`in modes`**

```
device Monitor
 features
  voltage: in data port real;
  alert: out data port bool;
end Monitor;

device implementation Monitor.Imp
 flows
  alert := (voltage < 4.5);
end Monitor.Imp;
```

Figure 3: Specification of the monitor.

clause, which can be declared along with port connections and subcomponents. This is demonstrated by the specification in Figure 2, which shows the usage of the battery component in the context of a redundant power system. It contains two instances of the battery device, being respectively active in the `primary` and the `backup` mode. The mode switch that initiates reconfiguration is triggered by an `empty` event arriving from the battery that is currently active.

The behaviour of a component after a re-activation (that is, an activation following a previous de-activation) depends on the definition of its starting mode. If the latter is declared using the **`initial`** attribute (such as mode `primary` of the `Power` component in Figure 2), then *mode history* is supported, that is, after re-activation the component resumes its operation without changing its mode or the values of its data elements. In contrast, the **`activation`** attribute (which is, e.g., attached to mode `charged` in Figure 1) indicates that each re-activation resets the component to its starting mode, using the default values for its data elements. In other words, the model assumes that batteries will be recharged upon re-activation.

Moreover, the definition of the power system includes a monitor component which checks the current voltage level and raises an alarm if it falls below a critical threshold of 4.5 [volts]. Its specification is shown in Figure 3; it employs another modelling concept, a so-called *flow*. A flow establishes a direct dependency between an outgoing data port of a component and (some of) its incoming data ports, meaning that a value update of one of the given incoming data ports immediately causes a corresponding update of the outgoing data port. In our concrete example, the flow-defined value of the outgoing data port `alert` is forwarded to the environment (by the data port connection `mon.alert -> alert` as defined in Figure 2) and is also used for triggering battery resets (by the connection to data port `tryReset` of the currently active battery).

2.2 Error Behaviour

Error models in SLIM are an extension to the specification of nominal models and are used to conduct safety and dependability analyses. For modularity, they are defined separately from nominal specifications. Akin to nominal models, an error model is defined by its type and its associated implementation.

An error model *type* defines an interface in terms of error states and (incoming and outgoing) error propagations. Error *states* are employed to represent the current configuration of the component with respect to the occurrence of errors. Error *propagations* are used to exchange error information between components.

```
error model BatteryFailure
 features
  ok: activation state;
  dead: error state;
  resetting: error state;
  batteryDied: out error propagation;
end BatteryFailure;

error model implementation BatteryFailure.Imp
 events
  fault: error event occurrence poisson 0.001;
  works: error event occurrence poisson 0.2;
  fails: error event occurrence poisson 0.8;
 transitions
  ok -[fault]-> dead;
  dead -[batteryDied]-> dead;
  dead -[reset]-> resetting;
  resetting -[works]-> ok;
  resetting -[fails]-> dead;
end BatteryFailure.Imp;
```

Figure 4: An error model.

An error model *implementation* provides the structural details of the error model. It is defined by a (probabilistic) machine over the error states declared in the error model type. Transitions between states can be triggered by error events, reset events, and error propagations.

Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations, and they can be annotated with occurrence distributions to model probabilistic error behaviour. Moreover, *reset* events can be sent from the nominal model to the error model of the same component, trying to repair a fault which has occurred. Whether or not such a reset operation is successful has to be modelled in the error implementation by defining (or respectively omitting) corresponding state transitions. Outgoing *error propagations* report an error state to other components. If their error states are affected, the other components will have a corresponding incoming propagation.

Figure 4 presents a basic error model for the battery device. It defines a probabilistic error event, `fault`, which occurs once every 1000 time units on average. Whenever this happens, the error model changes into the `dead` state. In the latter, the battery failure is signalled to the environment by means of the outgoing error propagation `batteryDied`. Moreover, the battery is enabled to receive a **`reset`** event from the nominal model to which the error behaviour is attached. It causes a transition to the `resetting` state, from which the battery recovers with a probability of 20%, and returns to the `dead` state otherwise.

Just as for nominal component specifications, we distinguish between **`initial`** and **`activation`** starting states. Their meaning is similar to that of initial and activation modes: if an initial state is given, the error model is put in that state only in the beginning of system execution, supporting *error history* during deactivation phases. With an activation state, the error model starts over again in that state after each (re-)activation of the respective component. This distinction is useful, e.g., for modelling the different error behaviour of hardware and software components: while reactivating a hardware component

(like a processor) will generally not remove the cause of an error, this is usually the case for software components (such as processes).

2.3 Fault Injection

As error models bear no relation with nominal models, an error model does not influence the nominal model unless they are linked through *fault injection*.

A fault injection describes the effect of the occurrence of an error on the nominal behaviour of the system. More concretely, it specifies the value update that a data element of a component implementation undergoes when its associated error model enters a specific error state.

To this aim, each fault injection has to be given by the user by specifying three parts: a state s in the error model, an outgoing data port or subcomponent d in the nominal model, and the fault effect given by the expression a. Multiple fault injections between error models and nominal models are possible.

The automatic procedure that integrates both models and the given fault injections, the so-called *model extension*, works as follows. The principal idea is that the nominal and error models are running concurrently. That is, the state space of the extended model consists of pairs of nominal modes and error states, and each transition in the extended model is due to a nominal mode transition, an error state transition, or a combination of both (in case of a reset operation). The above fault injection becomes enabled whenever the error model enters state s. In this case the assignment $d := a$ is carried out, i.e., the data subcomponent is assigned with the fault effect. This error effect is maintained as long as the error model stays in state s, overriding possible assignments to d in the nominal model. When s is left, the fault injection is disabled (but possibly another one is enabled).

For the power system example, the nominal model `Power.Imp` from Figure 1 and the error model `BatteryFailure.Imp` from Figure 4 can e.g. be linked through the injection of `voltage := 0.0` upon entering error state `dead`. The corresponding extended model is shown in Figure 5.

As the nominal and the error models contain two modes and three states, respectively, the extended model has six modes which are represented in the form $m\#s$ (for nominal mode m, and error state s). The transitions of the extended model fall into five different categories:

1. Nominal transitions are just copied if the error model is in its starting state (`ok`).
2. Error event transitions give rise to internal, probabilistic transitions which include the failure effects, if applicable.
3. Failure effects are also attached to nominal transitions if the error model is in the respective error state (`dead` in our example).
4. A combined reset transition is possible if it is enabled in both the nominal and the error model.
5. Finally, the outgoing error propagation `batteryDied` is turned into an event port, and corresponding transitions are added.

3 The COMPASS Toolset

The COMPASS toolset is the result of a significant implementation effort carried out by the COMPASS Consortium in a time frame of nearly three years. The design and implementation involved a full team of six researchers and twelve programmers. The GUI of the tool and most subcomponents (including the parsing routines, property manager, fault injection, and translators to and from SLIM) are implemented in the Python programming language, using the PyGTK library. Pre-existing components, such as the

```
device Battery
 features
  empty: out event port;
  tryReset: in data port bool default false;
  voltage: out data port real default 6.0;
  batteryDied: out event port;
end Battery;

device implementation Battery.Imp
 subcomponents
  energy: data continuous default 1.0;
 modes
  charged#ok: activation mode while energy' = -0.02 and energy >= 0.2;
  charged#dead, charged#resetting:
    mode while energy' = -0.02 and energy >= 0.2;
  depleted#ok, depleted#dead, depleted#resetting:
    mode while energy' = -0.03 and energy >= 0.0;
 transitions
   -- (i) Purely nominal transitions
  charged#ok -[then voltage := 2.0 * energy + 4.0]-> charged#ok;
  charged#ok -[empty when energy = 0.2]-> depleted#ok;
  depleted#ok -[then voltage := 2.0 * energy + 4.0]-> depleted#ok;
  -- (ii) Error event transitions
  charged#ok -[prob 0.001 then voltage := 0.0]-> charged#dead;
  depleted#ok -[prob 0.001 then voltage := 0.0]-> depleted#dead;
  charged#resetting -[prob 0.2]-> charged#ok;
  depleted#resetting -[prob 0.2]-> depleted#ok;
  charged#resetting -[prob 0.8 then voltage := 0.0]-> charged#dead;
  depleted#resetting -[prob 0.8 then voltage := 0.0]-> depleted#dead;
  -- (iii) Nominal transitions with fault injection
  charged#dead -[then voltage := 0.0]-> charged#dead;
  charged#dead -[empty when energy = 0.2]-> depleted#dead;
  depleted#dead -[then voltage := 0.0]-> depleted#dead;
  -- (iv) Reset transitions
  charged#dead -[when tryReset]-> charged#resetting;
  depleted#dead -[when tryReset]-> depleted#resetting;
  -- (v) Error propagation transitions
  charged#dead -[batteryDied]-> charged#dead;
  depleted#dead -[batteryDied]-> depleted#dead;
end Battery.Imp;
```

Figure 5: Battery component after model extension.

NuSMV and MRMC model checker, are instead written in C. Overall, the core of the COMPASS toolset consists of about 100,000 lines of Python code. Figure 6 visualizes the embedding of the toolset in the

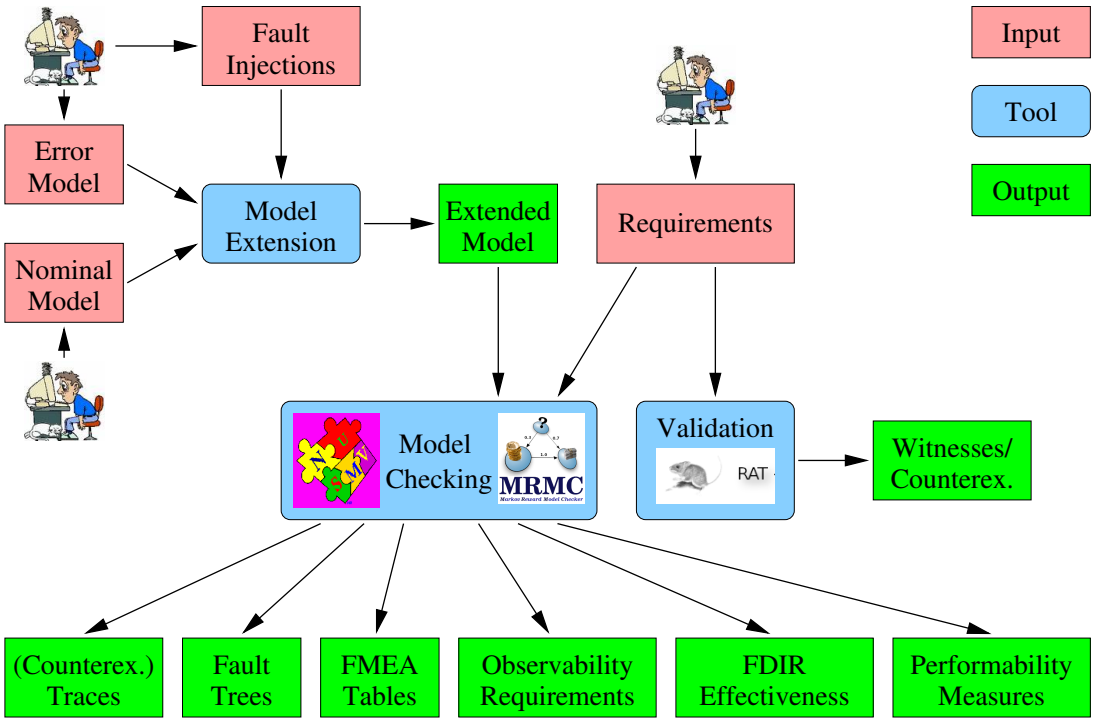


Figure 6: Methodology of the COMPASS Approach.

overall methodology.

The toolset takes as input one or more models written in the SLIM language, and a set of property patterns [16, 19]. Property patterns are internally converted into temporal logic before being fed to the underlying model checkers. This approach is intended to provide a simpler and better usable interface for system and software engineers, hiding the details of the temporal logic encoding. The COMPASS toolset provides templates for the most frequently used patterns. The tool generates several artefacts as output, among them: traces resulting either from simulation of the SLIM specification or as counterexamples for properties not satisfied by the specification; (probabilistic) fault trees and FMEA tables; diagnosability and performability measures. The toolset builds upon the following main components:

NuSMV [30, 13] (New Symbolic Model Verifier) is a symbolic model checker, supporting state-of-the-art verification techniques such as BDD-based and SAT-based verification techniques for CTL and LTL.

MRMC [29, 25, 27] (Markov Reward Model Checker) is a probabilistic model checker, supporting the analysis of discrete-time and continuous-time Markov reward models. It supports PCTL (Probabilistic Computation Tree Logic) and CSL (Continuous Stochastic Logic [2], a probabilistic real-time version of CTL), as well as variants thereof referring to expected costs and total accumulated costs.

SigRef [34] is a tool for minimising, amongst others, Interactive Markov Chains (IMC) with respect to different notions of bisimulation. It works in a fully symbolic manner using multi-terminal BDD representations of IMCs and applies signature-based algorithms.

Other components are used to manage parsing of models, parsing and editing of properties, fault injection and model extension, the translation from and to different languages supported by the verification

engines, and the visualization of the results.

In the remainder of this section, we overview the formal analyses supported by the COMPASS toolset.

3.1 Functional Correctness

The COMPASS toolset supports both traditional methods for analysis of functional correctness, such as simulation, and automated techniques for property verification, based on model checking.

Model-based simulation can be performed in two variants: random simulation, and guided simulation. Guided simulation comes in two different flavours: in guided-by-transitions simulation, the user can execute a step-by-step-simulation by choosing the next transition to be taken, among the enabled ones; in guided-by-values simulation, the user can perform a step-by-step-simulation by choosing a target value for one or more variables, and let the toolset choose a transition that drives the system into the target state, if any exists. The generated traces can be inspected using a trace manager that displays the values of the model variables of interest (filtering is possible) for each step, in a human-readable form; in case of timed and hybrid systems, timed transitions are highlighted.

Property verification is carried out by using model checking [3]. Model checking is an automated technique that verifies whether a property, typically expressed in temporal logic, holds for a given model. Symbolic techniques are used to tackle the state explosion problem. The COMPASS platform integrates the model checking capabilities provided by the NuSMV model checker. The version of NuSMV integrated in the COMPASS toolset supports both state-of-the-art BDD-based and SAT-based verification for finite-state systems, and SMT-based verification techniques for timed and hybrid systems, based on the MathSAT solver [6, 28]. On refutation of a property, a counterexample is generated by the model checker, showing an execution trace of the model violating the property; the trace can be inspected using the trace manager.

In addition to property model checking, it is also possible to run *deadlock checking*, in order to pinpoint deadlocks (states with no outgoing transitions) in the model, if there are any.

3.2 Safety Assessment

Model-based safety assessment of SLIM models aims at reducing the effort involved in safety assessment and at increasing the quality of the results by focusing on building formal models of the system, rather than carrying out the analyses. The analyses are based on symbolic model checking techniques [18, 10] first developed in the ESACS [17] and ISAAC [24] projects. As advocated in [11], an essential step to enable safety analysis is the decoupling between the nominal behaviour and the error behaviour of the system, realised by means of model extension, as explained in Section 2.3.

COMPASS automates traditional techniques for failure analysis, namely *Failure Mode and Effects Analysis* (FMEA) and *Fault Tree Analysis* (FTA). FMEA is an inductive (bottom-up) technique that starts by identifying a set of failure modes and, using forward reasoning, assesses their impact on a set of events (system properties). It requires a set of failure modes and a set of events to be analysed as input. FMEA typically considers single faults, but fault configurations involving several faults can be investigated as well. The set of fault configurations to be analysed is provided by specifying the cardinality of the FMEA table to be generated (generating a table of cardinality k amounts to considering sets of failure modes of cardinality *at most* k). The analysis results are summarised in an *FMEA table*, which links the given fault configurations with their effect on the given events. It is also possible to generate *dynamic* FMEA tables, namely to enforce an order of occurrence between failure modes within a fault configuration.

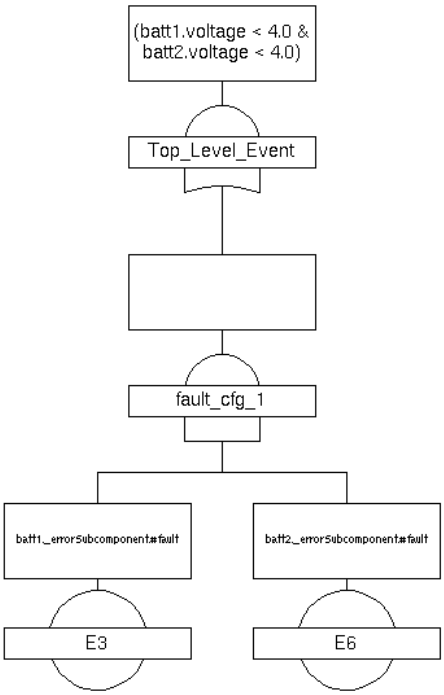


Figure 7: Example fault tree of the battery power system.

FTA is a deductive (top-down) technique, which, given a *top-level event* (TLE), i.e., the specification of an undesired condition, constructs all possible chains of one of more basic faults that contribute to the occurrence of the TLE, and pictorially represents these causal dependencies in a *fault tree*. In the COMPASS toolset, a fault tree can be represented as the collection of its minimal cut sets [11]. In this form, a fault tree has a two-layer logical structure, consisting of the disjunction of the minimal cut sets, each cut set being the conjunction of the corresponding basic faults. COMPASS also supports the generation of dynamic fault trees by analysing for each minimal cut set in which order the constituent basic faults have to occur to trigger the TLE; pictorially, such ordering constraints are represented using a priority AND (PAND) logical gate in the generated fault tree.

Figure 7 depicts a simple fault tree that is generated by FTA considering the TLE "`batt1.voltage < 4.0` **`and`** `batt2.voltage < 4.0`" in the power system model of Section 2. The fault tree contains only one minimal cut set (in the terminology introduced earlier in this section), corresponding to the unique child of the top-level OR gate, called *fault_cfg_1*. This cut set is a conjunction (AND gate in the fault tree) of two basic faults, namely `batt1._errorSubcomponent.#fault` and `batt2._errorSubcomponent.#fault`. The fault tree shows that the only cause that can lead to the occurrence of TLE is when both batteries die.

3.3 Diagnosability and FDIR Analysis

The COMPASS toolset also supports diagnosability analysis and FDIR (Fault Detection, Isolation and Recovery) effectiveness analysis. These analyses work under the hypothesis of partial observability. Rarely real applications are fully observable: parts of their state are hidden, and sensors are used to expose (partial) information about otherwise unobservable aspects. Diagnosis starts from the observed

run time behaviour of a system, and tries to provide an explanation (in terms of hidden states). Variables and ports in SLIM models can be declared to be observable, such as the outgoing data port `alert` in Figure 2.) The observables specify which variables or states are visible during execution. Diagnosis amounts to identifying the set of possible causes of a specific unexpected or faulty behaviour.

Diagnosability analysis is typically carried out before a fault management subsystem is available. It investigates the possibility for an ideal diagnosis system to infer accurate and sufficient run-time information on the behaviour of the observed system. Given a diagnosability property, diagnosability depends on the observed system and the available observations. The COMPASS toolset follows the approach described in [14]. In this framework, the violation of a diagnosability condition (e.g., fault detection) is reduced to the search of *critical pairs*. A critical pair is a pair of executions that are indistinguishable (i.e., they share the same inputs and observations), but hide conditions that should be distinguished (e.g., the fault is triggered only in one of the two executions). As an example, consider the property "`batt1 .voltage < 4.0 and batt2.voltage < 4.0`" in the power system model. This property turns out to be not diagnosable, with the set of observables provided in the model, that is, the signal `alert`. The reason for non-diagnosability lies in the fact that the signal alert does not allow to distinguish the case where the batteries die from the case where they are depleted (in mode `depleted`, the voltage may reach a minimum value of 4.0, whereas the signal `alert` is activated when the voltage decreases below 4.5). If we add another observable signal, say `alert2` defined as "`alert2 := (voltage < 4.0)`", then the system becomes diagnosable. Diagnosability analysis is implemented using model checking techniques that search for critical pairs in the so-called *twin plant* model which consists of a pair of instances of the original model, running in parallel. Using techniques similar to those used for computing minimal cut sets, it is also possible to automatically synthesise a set of observables that ensure diagnosability of a given SLIM model, see [4].

FDIR effectiveness analysis is carried out on an existing fault management subsystem, in order to ensure that it meets the diagnosability requirements. Fault detection is concerned with detecting whether a given system is malfunctioning. It checks whether a candidate observable signal can be considered as a *fault detection means* for a given fault, i.e., every occurrence of the fault will eventually cause the observable to be true. The COMPASS toolset reports all such observables as possible detection means. As an example, the observable `alert` is a fault detection means for the property "`batt1.voltage < 4.0 and batt2.voltage < 4.0`", as it will be activated whenever this property holds. Fault isolation analysis is concerned with identifying the specific cause of malfunctioning. It generates a fault tree that contains the minimal explanations that are compatible with the observable being true. In case of perfect isolation, the fault tree contains a single cut set consisting of one fault, indicating that the fault has been identified as the cause of the malfunctioning. A fault tree with more than one cut set indicates that there may be several explanations for the malfunctioning. As an example, the observable `alert` has two possible explanations in terms of elementary faults: either `batt1` has died, or `batt2` has died. Finally, fault recovery analysis is used to check whether a system is able to recover from a fault, according to a user-specified recoverability property. For instance, consider the property "`always (batt1.voltage < 4.4 implies eventually batt1.voltage > 5.5)`". This property states that whenever the voltage of `batt1` decreases below value 4.4, it will eventually recover, that is, it will be recharged so that its voltage is above value 5.5. This property is true in the nominal model, because the model assumes that batteries will be recharged upon re-activation (note that mode `charged` is declared to be an activation mode in Figure 1). However, this property does not hold when error behaviour is taken into account, as a battery may die, and the fault injection "`voltage := 0.0`" overrides the nominal behaviour in such a case (note that the reset operation may be unsuccessful). As for safety analysis, FDIR effectiveness analysis are implemented using model checking techniques.

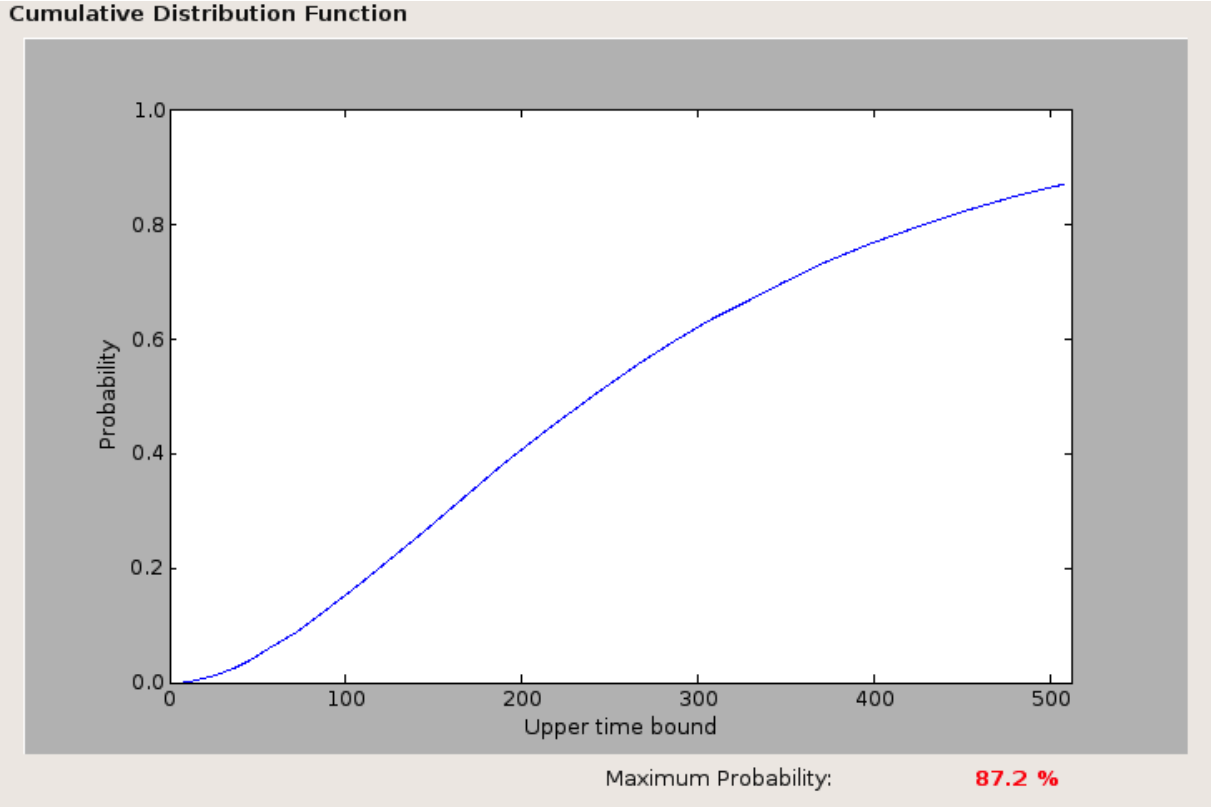


Figure 8: Cumulative distribution function curve resulting from performability analysis.

3.4 Performability Analysis

In order to guarantee the required performance, a SLIM model can be evaluated using probabilistic model checking techniques [3, Ch. 10]. The COMPASS toolset supports model checking of properties expressed in Continuous Stochastic Logic (CSL), which allows for the formal specification of well-known steady-state and transient probabilities, and more intricate performance measures such as combinations thereof.

As an example, consider a version of the redundant battery model in Section 2 where the energy is not expressed by a continuous differential equation, but by a discretised representation. This is needed for performability analysis, as it is yet unknown how to analyse stochastic hybrid systems with continuous-time stochastic aspects. Typical performance requirements of interest would be "the probability that the first battery dies within 100 hours" or "the probability that both batteries die within the mission duration".

In order to verify these performance requirements, the extended SLIM model undergoes several transformation steps. First, its reachable state space is generated through exhaustive (symbolic, i.e., BDD-based) exploration. In this phase, the error transitions, like those marked with `prob` in Figure 5, are viewed as ordinary transitions. In the second phase, the probabilistic rates (captured by `prob`) are interwoven through the previously generated state space by replacing the transition label with the associated probabilistic rate. The resulting state space is an Interactive Markov Chain, i.e., a Continuous-Time Markov Chain (CTMC) that may exhibit non-determinism [23]. This IMC is passed through the third phase, in which it is minimized using weak bisimulation minimization [15]. The IMC may turn into a

CTMC. Note that transitions in the resulting CTMC have rates that are either the same as the rates defined by the `occurrences` definition in the SLIM error model, or are, due to probabilistic bisimilarity, sums of those rates. Furthermore, the state labellings in that CTMC reflect the atomic propositions induced from the performance requirements. In the final phase the CSL formulae are extracted from the performance requirements using the patterns from [20] and then together with the CTMC fed to MRMC probabilistic model checker [25, 26, 29], to compute the desired probabilities. The result is a graph showing the cumulative distribution function over the time horizon specified in the performance requirement. An example screenshot of this from the COMPASS toolset is shown in Figure 8.

The same probabilistic model checking techniques are used for fault tree evaluation, i.e., computing the probability of the top-level event in dynamic fault trees by transforming it into its underlying Markov Chain [5].

References

[1] The COMPASS project web site. `http://compass.informatik.rwth-aachen.de/`.

[2] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. on Software Engineering*, 29(6):524–541, 2003.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] B. Bittner, M. Bozzano, A. Cimatti, and X. Olive. Symbolic synthesis of observability requirements for diagnosability. In *Proc. 26th AAAI Conf. on Artificial Intelligence*, pages 712–718. AAAI Press, 2012.

[5] H. Boudali, P. Crouzen, and M. Stoelinga. Dynamic fault tree analysis using input/output interactive Markov chains. In *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, pages 708–717. IEEE CS Press, 2007.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. Mathsat: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35:265–293, 2005.

[7] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The compass approach: Correctness, modelling and performability of aerospace systems. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, volume 5775 of *Lecture Notes in Computer Science*, pages 173–186. Springer, 2009.

[8] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended aadl models. *Computer Journal*, 54(5):754–775, 2011.

[9] M. Bozzano, A. Cimatti, M. Roveri, J.-P. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: a component-based modeling language. In *Proceedings of the 7th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 121–130. IEEE, 2009.

[10] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, editors, *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2007.

[11] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *Int. J. on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[12] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP – application to the verification of real-time systems. In *Models in Software Engineering, Workshops and Symposia at MODELS 2008*, volume 5421 of *LNCS*, pages 39–53. Springer-Verlag, 2008.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. J. on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[14] A. Cimatti, C. Pecheur, and R. Cavada. Formal verification of diagnosability via symbolic model checking. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 363–369. Morgan Kaufmann, 2003.

[15] S. Derisavi, H. Hermanns, and W. Sanders. Optimal state-space lumping in Markov chains. *Information Processing Letters*, 87(6):309–315, 2003.

[16] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 411–420. ACM, 1999.

[17] ESACS Project. `http://www.esacs.org`.

[18] FSAP/NuSMV-SA Platform. `http://sra.fbk.eu/tools/FSAP`.

[19] L. Grunske. Specification patterns for probabilistic quality properties. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 31–40. ACM, 2008.

[20] L. Grunske. Specification patterns for probabilistic quality properties. In *Int. Conf. on Software Engineering (ICSE)*, pages 31–40. ACM, 2008.

[21] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[22] T. Henzinger. The theory of hybrid automata. In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 278–292. IEEE CS Press, 1996.

[23] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer, 2002.

[24] ISAAC Project. `http://www.isaac-fp6.org`.

[25] J.-P. Katoen, M. Khattri, and I. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE CS, 2005.

[26] J.-P. Katoen, I. Zapreev, E. M. Hahn, H. Hermanns, and D. Jansen. The ins and outs of the probabilistic model checker MRMC. In *Quantitative Evaluation of Systems (QEST)*, pages 167–176. IEEE CS Press, 2009.

[27] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, 68(2):90–104, 2011.

[28] MathSAT. `http://mathsat.fbk.eu`.

[29] MRMC – The Markov Reward Model Checker. `http://www.mrmc-tool.org/`.

[30] NuSMV Model Checker. `http://nusmv.fbk.eu`.

[31] Architecture Analysis and Design Language (AADL), rev. a. SAE Standard AS5506A, International Society of Automotive Engineers, Jan. 2009.

[32] Architecture Analysis and Design Language (AADL) V2. SAE Draft Standard AS5506 V2, International Society of Automotive Engineers, Mar. 2008.

[33] Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1, International Society of Automotive Engineers, June 2006.

[34] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref - a symbolic bisimulation tool box. In S. Graf and W. Zhang, editors, *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2006.

Technical Talks

Runtime Verification

Martin Leucker

Institut für Informatik, TU München, München, Germany

Starting from a definition of runtime verification, we develop a taxonomy that explains the different aspects of runtime verification. We explain the core idea of runtime verification by showing how monitors can be attached to existing programs, be used to verify certain aspects of the underlying program as well as be used to guide the program execution. The main part of the presentation deals with synthesis techniques that, starting from a high level correctness specifications, derive suitable monitors automatically. We start with properties expressed in linear temporal logic (LTL), first with a semantics on finite traces and then extended to a semantics over infinite traces.

Computer-Aided Cryptographic Proofs and Designs

Gilles Barthe

IMDEA Software, Madrid, Spain

EasyCrypt is a tool for constructing and verifying cryptographic proofs. EasyCrypt can be used as a stand-alone application, or as a verifying back-end for cryptographic compilers. SyntheCrypt is a new tool that synthesizes public-key encryption schemes and generates proofs of security in EasyCrypt. The presentation will outline the language-based methods that underlie the design of both tools and illustrate some of their applications.

Static Analysis by Abstract Interpretation of Sequential and Multithreaded Programs

Antoine Miné*
CNRS & cole Normale Suprieure 45, rue d'Ulm 75005 Paris, France
mine@di.ens.fr

Abstract

In the realm of embedded critical systems, it is crucial to guarantee the correctness of programs before they are deployed. Static analyzers can help by detecting at compile-time potentially erroneous program behaviors: they perform sound over-approximations to achieve an efficient analysis while not missing any potential behavior. We discuss the systematic design of such analyzers using abstract interpretation, a general theory of semantic approximation. After recalling the classic construction of static analyzers for sequential programs by abstraction of the concrete trace semantics, we introduce abstractions to derive thread-modular analyzers for multithreaded programs, borrowing ideas from rely/guarantee proof methods. Finally, we present two static analyzer tools, Astre and AstreA, that are used to check for run-time errors in large sequential and multithreaded embedded industrial avionic C applications.

1 Introduction

It is crucial to guarantee the correctness of programs before they are deployed, especially in the realm of embedded critical systems, where software cannot be corrected during missions and a single mistake can have dramatic consequences. Testing, the most widespread verification method employed in industry, is very efficient at catching errors, but it is costly and cannot, for efficiency reasons, test all executions. Hence, testing can miss errors. This is especially true for parallel and multithreaded programs: the huge number of possible thread interleavings causes a combinatorial explosion of executions, while some errors only appear in a tiny fraction of them (such as data-races). Formal methods, on the other hand, can provide rigorous guarantees that all the executions are correct.

We consider here static analyses, able to inspect program sources in order to find defects. We only consider semantic analyses, that are based on a mathematical notion of program behaviors, as opposed to syntax-based style checkers. Unlike proof methods, which require the user to provide annotations, these analyses run on the original, unannotated source without human intervention. Full automation and efficiency imply that such analyses are approximated. We impose soundness: unlike testing, program behaviors are over-approximated so that, if an error is present, it will be detected. However, the analysis can report spurious errors. Sound static analyzers have been used for decades in applications, such as program optimization, where precision is not critical. Recent progress had lead to the design of tools able to check for simple but important safety properties (such as the absence of run-time error) with few or zero false alarms. We participated in the design of two of them [2]: Astre, an industrial-strength analyzer

*This work was partially supported by the INRIA project "Abstraction" common to CNRS and ENS in France, and by the project ANR-11-INSE-014 from the French *Agence nationale de la recherche*.

that checks for run-time errors in synchronous embedded C code, and AstreA, its prototype extension targeting multithreaded embedded C applications.

These analyzers are based on abstract interpretation, which is a general theory of program semantics introduced by Cousot and Cousot in the late '70s [5]. Abstract interpretation stems from the observation that many semantics can be uniformly expressed as fixpoints of operators, after which seemingly unrelated semantics can be compared. It expresses formally the fact that some semantics are more abstract than (lose information with respect to) others as they compute approximations in less rich domains of properties. It provides tools to prove soundness: any property proved in an abstract semantic is still true in the concrete one. Finally, it provides tools to design and combine abstractions.

The aim of the article is to give a short overview of the theory underlying tools such as Astre and AstreA. Section 2 focuses on sequential programs and presents the classic construction of an effective analyzer by abstraction of the most concrete semantics of a program: its execution traces. Section 3 considers multithreaded programs and explains how, borrowing ideas from rely/guarantee proof methods [14], an efficient, thread-modular analysis can be constructed. Section 4 briefly presents the Astre and AstreA analyzers, and Sec. 5 concludes.

The modest contribution of this article is the formulation, in Sec. 3.4, of Jones' rely/guarantee method [14] in abstract interpretation form, as an abstraction of the execution traces of multithreaded programs by decomposition into intra-thread invariants and inter-thread interferences. This extends previous work in the 80's [7] that formalized earlier Owicki–Gries–Lamport methods [17, 15] as abstract interpretation. It also extends our recent previous work [16], that only considered coarse abstract interferences, by exhibiting an intermediate layer of abstraction from which it can be recovered.

2 Abstractions for Sequential Programs

As a short introduction to abstract interpretation concepts, we review the formal construction of a static analyzer for *sequential* programs by abstraction of its trace semantics — see also [2, § II] for an extended introduction.

2.1 Transition Systems

The semantics of a program is defined classically [5] in a very general way in *small step operational form*, as a *transition system*: (Σ, τ, I), where Σ is a set of states, $I \subseteq \Sigma$ is the subset of initial states, and $\tau \subseteq \Sigma \times \Sigma$ is a transition relation. For sequential programs, the state set is defined as $\Sigma \stackrel{\text{def}}{=} \mathscr{L} \times \mathscr{M}$, where each state $\sigma = (\ell, m)$ has a control part $\ell \in \mathscr{L}$ denoting the current program point, and a memory part $m \in \mathscr{M} \stackrel{\text{def}}{=} \mathscr{V} \to \mathbb{V}$ mapping program variables $V \in \mathscr{V}$ to values $v \in \mathbb{V}$. A transition models an atomic execution step, such as executing a machine-code instruction in a program. We denote by $(\ell, m) \to (\ell', m')$ the fact that $((\ell, m), (\ell', m')) \in \tau$, i.e, the program can reach the state (ℓ', m') from the state (ℓ, m) after one execution step. The transition system derives mechanically from the program code itself, e.g., from the sequence of binary instructions or, more conveniently, by induction on the syntax tree of the source code.

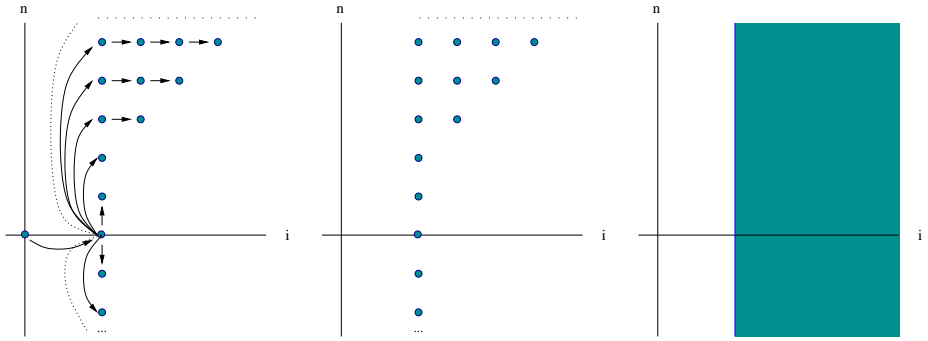
Example 1. Consider the simple programming language syntax:

$$ {}^{\ell}P^{\ell'} ::= {}^{\ell}x := e^{\ell'} \mid {}^{\ell}\texttt{if}\ e\ \texttt{then}\ {}^{\ell''}P^{\ell'} \mid {}^{\ell}\texttt{while}\ e\ \texttt{do}\ {}^{\ell''}P^{\ell'} \mid {}^{\ell}P;{}^{\ell''}P'^{\ell'} $$

```
(1)  i := 2;                                  at (1): i = 0 ∧ n = 0
(2)  n := input_int();                        at (2): i = 2 ∧ n = 0
(3)  while i < n do                           at (3): 2 ≤ i ≤ max(2,n)
(4)      if input_bool() then i := i + 1;     at (4): 2 ≤ i ≤ n − 1 ∧ n ≥ 3
(5)  done;                                    at (5): 2 ≤ i ≤ n ∧ n ≥ 3
(6)  assert i >= 2;                           at (6): i = max(2,n)
```

(a) (b)

Figure 1: A simple sequential program (a), and invariant assertions (b).



(a) (b) (c)

Figure 2: Semantics of the program in Fig. 1 at various levels of abstractions: (a) traces, and (b) reachable states and (c) intervals at program point (3).

where $^{\ell}$ and e denote respectively syntactic program points and expressions. The transition system $\tau[{}^{\ell}P^{\ell'}]$ is derived by induction on the syntax of P as follows:

$$
\begin{array}{lcl}
\tau[{}^{\ell}x := e^{\ell'}] & \stackrel{\text{def}}{=} & \{(\ell,m) \to (\ell',m[x \mapsto v]) \,|\, m \in \mathscr{M}, e \overset{m}{\rightsquigarrow} v\} \\
\tau[{}^{\ell}\texttt{if } e \texttt{ then } {}^{\ell''}P^{\ell'}] & \stackrel{\text{def}}{=} & \{(\ell,m) \to (\ell'',m) \,|\, m \in \mathscr{M}, e \overset{m}{\rightsquigarrow} \texttt{true}\} \cup \tau[{}^{\ell''}P^{\ell'}] \cup \\
 & & \{(\ell,m) \to (\ell',m) \,|\, m \in \mathscr{M}, e \overset{m}{\rightsquigarrow} \texttt{false}\} \\
\tau[{}^{\ell}\texttt{while } e \texttt{ do } {}^{\ell''}P^{\ell'}] & \stackrel{\text{def}}{=} & \{(\ell,m) \to (\ell'',m) \,|\, m \in \mathscr{M}, e \overset{m}{\rightsquigarrow} \texttt{true}\} \cup \tau[{}^{\ell''}P^{\ell}] \cup \\
 & & \{(\ell,m) \to (\ell',m) \,|\, m \in \mathscr{M}, e \overset{m}{\rightsquigarrow} \texttt{false}\} \\
\tau[{}^{\ell}P;{}^{\ell''}P'^{\ell'}] & \stackrel{\text{def}}{=} & \tau[{}^{\ell}P^{\ell''}] \cup \tau[{}^{\ell''}P'^{\ell'}]
\end{array}
$$

where $m[x \mapsto v]$ denotes the function mapping x to v and $y \neq x$ to $m(y)$, and $e \overset{m}{\rightsquigarrow} v$ states that e can evaluate to the value v in the memory m. ∎

2.2 Trace Semantics

We are not interested in the program itself, but in the properties of its executions. Formally, an execution *trace* is a finite sequence of states $\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n$ such that $\sigma_0 \in I$ and $\forall i, (\sigma_i, \sigma_{i+1}) \in \tau$. The semantics we want to observe, which is called a *collecting semantics*, is thus defined in our case as the set $T \in \mathscr{P}(\Sigma^*)$ of traces spawned by the program. It can be expressed [6] as the least fixpoint of a

continuous operator in the complete lattice of sets of traces:

$$\begin{array}{l} T = \mathrm{lfp}\, F \text{ where} \\ F(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma_0 \to \cdots \to \sigma_{i+1} \mid \sigma_0 \to \cdots \to \sigma_i \in X \wedge \sigma_i \to \sigma_{i+1} \,\} \end{array} \tag{1}$$

i.e., T is the smallest set containing traces reduced to an initial state and closed under the extension of traces by an additional execution step. Note that we are observing *partial* execution traces; informally, we allow executions to be interrupted prematurely; formally, T is closed by prefix (it also includes the finite prefixes of non-terminating executions). This is sufficient if we are interested in *safety* properties, i.e., properties of the form "no bad state is ever encountered," which is the case here — more general trace semantics, able to also reason about liveness properties, are discussed for instance in [4]. A classic result [6] is that T can be restated as the limit of an iteration sequence: $T = \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. It becomes clear then that computing T is equivalent to testing the program on all possible executions (albeit with an unusual exhaustive breadth-first strategy) and that it is not adapted to the effective and efficient verification of programs: when the program has unbounded or infinite executions, T is infinite.

Example 2. The simple program in Fig. 1.a increments `i` in a loop, until it reaches a user-specified value `n`. Figure 2.a presents its trace semantics starting in the state set $I = \{\, (1, [\mathtt{m} \mapsto 0, \mathtt{i} \mapsto 0]) \,\}$. The program has infinite executions (e.g., if `i` is never incremented). ∎

2.3 State Semantics

We observe that, in order to prove safety properties, it is not necessary to compute T exactly. It is sufficient to collect the set $S \in \mathscr{P}(\Sigma)$ of program states encountered, abstracting away any information available in T on the ordering of states. We use an *abstraction function* $\alpha^{state} : \mathscr{P}(\Sigma^*) \to \mathscr{P}(\Sigma)$:

$$S = \alpha^{state}(T) \text{ where } \alpha^{state}(X) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \to \cdots \to \sigma_n \in X, i \in [0;n], \sigma = \sigma_i \,\} \tag{2}$$

An important result is that, as T, S can be computed directly as a fixpoint:

$$S = \mathrm{lfp}\, G \text{ where } G(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma' \mid \exists \sigma \in X, \sigma \to \sigma' \,\}$$

or, equivalently, as an iteration sequence $S = \bigcup_{n\in\mathbb{N}} G^n(\emptyset)$, which naturally expresses that S is the set of states reachable from I after zero, one, or more transitions. The similarity in fixpoint characterisation of S and T is not a coincidence, but a general result of abstract interpretation (although, in most cases, the abstract fixpoint only over-approximates the concrete one: $\mathrm{lfp}\, G \supseteq \alpha^{state}(\mathrm{lfp}\, F)$, see [4]). Dually, given a set of states S, one can construct the set of traces abstracted by S using a *concretization* function $\gamma^{state} \stackrel{\text{def}}{=} \lambda S.\, \{\, \sigma_0 \to \cdots \to \sigma_n \mid n \in \mathbb{N} \wedge \forall i \in [0;n], \sigma_i \in S \,\}$. The pair $(\alpha^{state}, \gamma^{state})$ forms a Galois connection.[1] A classic result [5] states that the *best abstraction* of F can be defined as $\alpha^{state} \circ F \circ \gamma^{state}$, which in our case turns out to be exactly G. When the set of states is finite (e.g., when the memory is bounded), S can always be computed by iteration in finite time, even if T cannot. Obviously, S may be extremely large and require many iterations to stabilize, hence computing S exactly is not a practical solution; we will need further abstractions.

[1] I.e., $\forall X \in \mathscr{P}(\Sigma^*), \forall Y \in \mathscr{P}(\Sigma),\ \alpha^{state}(X) \subseteq Y \iff X \subseteq \gamma^{state}(Y)$.

2.4 Program Proofs and Inductive Invariants

There is a deep connection [5] between the state semantics and the program logic of Floyd–Hoare [13] used to prove partial correctness. If we interpret each logic assertion A_ℓ at program point ℓ as the set of memory states $[\![A_\ell]\!] \subseteq \mathscr{M}$ satisfying it, and note $M \stackrel{\text{def}}{=} \{(\ell,m) \,|\, m \in [\![A_\ell]\!]\}$, then $(A_\ell)_{\ell\in\mathscr{L}}$ is a valid (i.e., inductive) invariant assertion if and only if $G(M) \subseteq M$. Moreover, the best inductive invariant assertion stems from lfp G, i.e., it is S. While, in proof methods, the inductive invariants must be devised by an oracle (the user), abstract interpretation opens the way to automatic invariant inference by providing a *constructive* view on invariants (through iteration) and allowing further abstractions.

Example 3. The optimal invariant assertions of the program in Fig. 1.a appear in Fig. 1.b, and Fig. 2.b presents graphically its state abstraction at point (3). ∎

2.5 Memory Abstractions

In order gain in efficiency on bounded state spaces and handle unbounded ones, we need to abstract further. As many errors (such as overflows and divisions by zero) are caused by invalid arguments of operators, a natural idea is to observe only the set of values each variable can take at each program point. Instead of concrete state sets in $\mathscr{P}(\Sigma) \stackrel{\text{def}}{=} \mathscr{P}(\mathscr{L} \times (\mathscr{V} \to \mathbb{V}))$, we manipulate abstract states in $\Sigma_C \stackrel{\text{def}}{=} (\mathscr{L} \times \mathscr{V}) \to \mathscr{P}(\mathbb{V})$. The concrete and abstract domains are linked through the following Galois connection (so-called Cartesian Galois connection):

$$\begin{array}{lcl} \alpha^{cart}(X) & \stackrel{\text{def}}{=} & \lambda(\ell,V)\,.\,\{v \,|\, \exists (\ell,m) \in X, m(V) = v\} \\ \gamma^{cart}(X^\sharp) & \stackrel{\text{def}}{=} & \{(\ell,m) \,|\, \forall V, m(V) \in X^\sharp(\ell,V)\} \end{array} \tag{3}$$

Assuming that variables are integers or reals, a further abstraction consists in maintaining, for each variable, its bounds instead of all its possible values. We compose the connection $(\alpha^{cart}, \gamma^{cart})$ with (the element-wise lifting of) the following connection $(\alpha^{itv}, \gamma^{itv})$ between $\mathscr{P}(\mathbb{R})$ and $(\mathbb{R} \cup \{-\infty\}) \times (\mathbb{R} \cup \{+\infty\})$:

$$\begin{array}{lcl} \alpha^{itv}(X) & \stackrel{\text{def}}{=} & [\min X; \max X] \\ \gamma^{itv}([a;b]) & \stackrel{\text{def}}{=} & \{x \in \mathbb{R} \,|\, a \leq x \leq b\} \end{array} \tag{4}$$

yielding the interval abstract domain [5].

Another example of memory domain is the linear inequality domain [10] that abstracts sets of points in $\mathscr{P}(\mathscr{V} \to \mathbb{R})$ into convex, closed polyhedra. One abstracts $\mathscr{P}(\Sigma) \simeq \mathscr{L} \to \mathscr{P}(\mathscr{V} \to \mathbb{V})$ by associating a polyhedron to each program point, which permits the inference of linear relations between variables. The polyhedra domain is thus a relational domain, unlike the domains deriving from the Cartesian abstraction (such as intervals).

Example 4. Figure 2.c presents the interval abstraction of the program in Fig. 1.a at point (3). The relational information that $i \leq n$ when $n \geq 2$ is lost. ∎

Following the same method that derived the state semantics from the trace one, we can derive an interval semantics from the state one: it is expressed as a fixpoint lfp $G^\sharp$ of an interval abstraction $G^\sharp$ of G. While it is possible to define an optimal $G^\sharp$ as $\alpha \circ G \circ \gamma$ — where (α,γ) combines $(\alpha^{itv}, \gamma^{itv})$ and $(\alpha^{cart}, \gamma^{cart})$ — this is a complex process when G is large, and it must be performed for each G, i.e., for each program. To construct a general analyzer, we use the fact that F, and so, $G = \alpha^{state} \circ F \circ \gamma^{state}$, are built by combining operators for atomic language constructs, as in Ex. 1. It is thus possible to derive $G^\sharp$ as a combination of a small fixed set of abstract operators. More precisely, we only have

to design abstraction versions of assignments, tests, and set union $\cup$. Generally, the combination of optimal operators is not optimal, and so, the resulting $G^\sharp$ is not optimal — e.g., $(\alpha \circ G_1 \circ G_2 \circ \gamma)(X) \subsetneq ((\alpha \circ G_1 \circ \gamma) \circ (\alpha \circ G_2 \circ \gamma))(X)$. Additionally, due to efficiency and practicality concerns, the base abstract operators may be chosen non-optimal to begin with. Achieving a sound analysis, i.e., $\gamma(\text{lfp } G^\sharp) \supseteq \text{lfp } G$, only requires sound abstracts operators, i.e., $\forall X^\sharp, (\gamma \circ G^\sharp)(X^\sharp) \supseteq (G \circ \gamma)(X^\sharp)$. The fact that $G^\sharp \neq \alpha \circ G \circ \gamma$, and even the existence of an α function, while sometimes desirable, is by no mean required — for instance, the polyhedra domain [10] does not feature an abstraction function α as some sets, such as discs, have no best polyhedral abstraction.

Example 5. The interval abstraction $[\![{}^\ell x := y + z^{\ell'}]\!]^\sharp$ of a simple addition maps the interval environment $[x \mapsto [l_x;h_x]; y \mapsto [l_y;h_y]; z \mapsto [l_z;h_z]]$ at point ℓ to $[x \mapsto [l_y + l_z; h_y + h_z]; y \mapsto [l_y;h_y]; z \mapsto [l_z;h_z]]$ at point ℓ', which is optimal. ∎

Example 6. The optimal abstraction $[\![{}^\ell y := -z; {}^{\ell'} x := y + z^{\ell''}]\!]^\sharp$ would map x to $[0;0]$. However, the combination $[\![{}^\ell y := -z^{\ell'}]\!]^\sharp \circ [\![{}^{\ell'} x := y + z^{\ell''}]\!]^\sharp$ maps x to $[l_z - h_z; h_z - l_z]$ instead, which is coarser when $l_z < h_z$. ∎

Example 7. We can design a sound non-optimal fall-back abstraction for arbitrary assignments ${}^\ell x := e^{\ell'}$ by simply mapping x to $[-\infty;+\infty]$. ∎

We now know how to define systematically a sound abstract operator $G^\sharp$ which can be efficiently computed. Nevertheless, the computation of lfp $G^\sharp$ by naive iteration may not converge fast enough (or at all). Hence, abstract domains are generally equipped with a convergence acceleration binary operator $\triangledown$, called *widening* and introduced in [5]. Widenings extrapolate between two successive iterates and ensure that any sequence $X^\sharp_{i+1} \overset{\text{def}}{=} X^\sharp_i \triangledown G^\sharp(X^\sharp_i)$ converges in finite time, possibly towards a coarse abstraction of the actual fixpoint.

Example 8. The classic interval widening [5] sets unstable bounds to infinity:

$$[l_1;h_1] \triangledown [l_2;h_2] \overset{\text{def}}{=} \left[\begin{cases} -\infty & \text{if } l_2 < l_1 \\ l_1 & \text{otherwise} \end{cases} ; \begin{cases} +\infty & \text{if } h_2 > h_1 \\ h_1 & \text{otherwise} \end{cases} \right] \tag{5}$$

When analyzing Fig. 1, the iterations with widening at (3) give the following intervals for `i`: $i^\sharp_0 = [2;2]$ and $i^\sharp_1 = [2;2] \triangledown [2;3] = [2;+\infty]$, which is stable. ∎

To balance the accumulation of imprecision caused by widenings and combining separately abstracted operators, it is often necessary to reason on an abstract domain strictly more expressive than the properties we want to prove — e.g., the polyhedra domain may be necessary to infer variable bounds that the interval domain cannot infer, although it can represent them, such as $x = 0$ in Ex. 6.

2.6 Further Considerations

We end this introduction with some pointers to additional material. Firstly, there exists a large library of abstract domains, in particular numeric ones, and associated operators with various precision versus cost trade-offs — [2] describes a few of them. An actual analyzer will use a combination, such as a reduced product, of many domains [9]. Secondly, we have assumed for simplicity that the set of variables $\mathscr{V}$ is finite, but abstract domains able to handle an unbounded memory exist; this is necessary when dynamic memory allocation is used. They often combine abstractions of the memory shape and of the contents of numeric fields, as in [19]. Likewise, an efficient handling of procedures requires abstracting the control state $\mathscr{L}$ (which can be large, or even unbounded in the case of recursivity). Call-string methods [18] are an instance of such abstractions. Finally, alternate iteration techniques exist [3], as well as methods to decompose the global fixpoint into several local ones to achieve a modular analysis [8].

3 Abstractions for Parallel Programs

We now consider multithreaded programs in a shared memory. We show how, starting from a classic concrete semantics based on interleaved executions [11] and applying abstract interpretation, we can construct an effective thread-modular static analysis which is similar to rely/guarantee proof methods [14].

3.1 Labelled Transition Systems

We assume a *finite* set $\mathscr{T}$ of threads. Each thread $t \in \mathscr{T}$ has its own control space $\mathscr{L}_t$ and transition relation $\tau_t \subseteq \Sigma_t \times \Sigma_t$, where $\Sigma_t \stackrel{\text{def}}{=} \mathscr{L}_t \times \mathscr{M}$. The state space and transitions for the whole program are derived from those of individual threads as follows. Program states live in $\Sigma \stackrel{\text{def}}{=} (\prod_{t \in \mathscr{T}} \{t\} \to \mathscr{L}_t) \times \mathscr{M}$, i.e., each thread has its own control point, but the memory is shared. The semantics of the program is defined as a *labelled* transition system (Σ, τ, I), where the transition relation $\tau \subseteq \Sigma \times \mathscr{T} \times \Sigma$ is defined as:[2] $((\bar{\ell}, m), t, (\bar{\ell}', m')) \in \tau$ if $((\bar{\ell}[t], m), (\bar{\ell}'[t], m)) \in \tau_t$ and $\forall t' \neq t, \bar{\ell}[t'] = \bar{\ell}'[t']$. It states that an execution step of the program is an execution step of some thread and updates only that thread's control location. We denote such a step as $(\bar{\ell}, m) \xrightarrow{t} (\bar{\ell}', m')$. Labels $t \in \mathscr{T}$ are used to explicitly remember which thread caused each transition.

3.2 Interleaved Trace Semantics

As for sequential programs, we consider finite prefixes of executions and ignore liveness properties — in the context of parallel programs, liveness properties are related to fairness conditions, which is a very complex issue not discussed here; see [11] for a complete treatment. The trace semantics T is defined as in (1):

$$\begin{array}{l} T = \mathrm{lfp}\, F \text{ where} \\ F(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \mid \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in X \wedge \sigma_i \xrightarrow{t_i} \sigma_{i+1} \,\} \end{array} \tag{6}$$

which thus corresponds to executions consisting of arbitrary *interleavings* of transitions from any threads.

Given the similarity with the sequential trace semantics, a natural idea to analyze a parallel program is to forget about labels and apply the state and memory abstractions (Secs. 2.3, 2.5). The problem with this method is the huge number of control states to consider: $\prod_{t \in \mathscr{T}} |\mathscr{L}_t|$ instead of $|\mathscr{L}|$. Moreover, τ is much larger that $\bigcup_{t \in \mathscr{T}} \tau_t$ as a single transition in τ_t is repeated for each possible control state of other threads $\prod_{t' \neq t} \mathscr{L}_{t'}$. As a consequence, constructing an abstraction $G^\sharp$ of F from abstractions of atomic instructions requires a combination of many more such functions than for sequential programs. This makes even coarse Cartesian interval abstractions impracticable.

3.3 Rely/Guarantee

Proof methods for parallel programs, such as Owicki–Gries–Lamport [17, 15] and rely/guarantee [14] solve this issue by attaching invariants to thread control points in $\bigcup_{t \in \mathscr{T}} \mathscr{L}_t$ instead of combinations of thread points in $\prod_{t \in \mathscr{T}} \mathscr{L}_t$. The price to pay is a refined notion of invariance: the properties attached to a thread t must also be proved stable by the effect of the other threads $t' \neq t$. In rely/guarantee methods, this effect is explicitly provided as extra annotations in t that are assumed to hold (*relied* on) when

[2] The $\bar{x}$ notations indicates that $\bar{x}$ is a finite vector. $\bar{x}[i]$ is its value at index i. We also use vector functions $\bar{f}$. The i-th component of y's image $\bar{f}(y)$ is denoted $\bar{f}(y)[i]$. Finally, $\bar{\emptyset}$, $\bar{\cup}$, and $\bar{\subseteq}$ denote respectively $\emptyset$, $\cup$, $\subseteq$ extended element-wise to vectors.

```
(1) while true do
(2)     if x < y then
(3)         x := x + 1;
(4) done;
```

```
(5) while true do
(6)     if y < 10 then
(7)         y := y + 1;
(8) done;
```

(a)

at (1),(2),(4):	$x \leq y$
at (3):	$x < y$
at (5),(6),(8):	$y \leq 10$
at (7):	$y < 10$

(b)

Figure 3: A simple multithreaded program (a), and its invariant assertions (b).

```
(1) while true do
(2)     x := x + 1;
(3)     x := x - 1;
(4) done;
```

```
(5) while true do
(6)     x := x - 2;
(7)     x := x + 2;
(8) done;
```

(a)

at (1),(2),(4):	$x \in \{-2, 0\}$
at (3):	$x \in \{-1, 1\}$
at (5),(6),(8):	$x \in \{0, 1\}$
at (7):	$x \in \{-2, -1\}$

(b)

Figure 4: A program requiring flow-sensitivity (a), and its invariant assertions (b).

checking the invariants for t and proved correct (*guaranteed*) when checking the other threads $t' \neq t$. It then becomes possible to check each thread in a modular way, i.e., without looking at the code of the other threads, relying on the annotations instead.

Example 9. Figure 3 presents a program with two threads: the first one increments x up to y and the second one increments y up to 10. For conciseness, we exemplify the rely/guarantee conditions at a single program point. Consider the problem of proving that $x < y$ holds at (3), just after the test. We need to prove that the assertion is stable by the action of the second thread. It is sufficient to *rely* on the fact that the second thread does not modify x and only increments y. This property is, in turn, *guaranteed* by analyzing the second thread, relying only on the fact that the first thread does not modify y. ∎

3.4 Interference Semantics

We now rephrase the idea of rely/guarantee methods as an abstract interpretation of the interleaved trace semantics T (6). We decompose the trace semantics using two complementary abstractions: an abstraction into *thread-local invariants* (which is inspired from the formalization in [7] of Owicki–Gries–Lamport methods [17, 15] as an abstract interpretation) and an abstraction into *inter-thread interferences* (which is new). Firstly, the local invariant $\bar{S}[t]$ of a thread t is defined by projecting, with the bijection π_t, the reachable states $\alpha^{state}(T)$ (2) on t's control state $\mathscr{L}_t$:

$$\bar{S}[t] \stackrel{\text{def}}{=} (\Pi_t \circ \alpha^{state})(T) \text{ where}$$
$$\Pi_t(X) \stackrel{\text{def}}{=} \{\, \pi_t(x) \,|\, x \in X \,\} \text{ and } \pi_t(\bar{\ell}, m) \stackrel{\text{def}}{=} (\bar{\ell}[t], m[\forall t' \neq t,\, p_{t'} \mapsto \bar{\ell}[t']])$$

We keep the control state $\bar{\ell}[t']$ of other threads $t' \neq t$ encoded as extra variables $p_{t'}$ in the memory (called *auxiliary variables* in Owicki–Gries [17]). It is possible to remove these variables and keep only t's control information to get:

$$\alpha_t^{ctrl}(\bar{S}[t]) \text{ where } \alpha_t^{ctrl}(X) \stackrel{\text{def}}{=} \{\, (\ell, m) \,|\, \exists \bar{\ell},\, \pi_t(\bar{\ell}, m) \in X \wedge \bar{\ell}[t] = \ell \,\} \tag{7}$$

but α_t^{ctrl} is known to be an incomplete abstraction: some invariance properties on $\bar{S}[t]$ cannot be proved using $\alpha_t^{ctrl}(\bar{S}[t])$ only (see Ex. 12). Secondly, the interference $\bar{A}[t]$ of a thread t is defined as the possible

actions it has on other threads, as observed in the interleaved trace semantics T:

$$\bar{A}[t] \stackrel{\text{def}}{=} \{ (\sigma_i, \sigma_{i+1}) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T, t_i = t \} \tag{8}$$

Interferences are actually subsets of the transition relation *restricted* to those transitions appearing in actual traces starting in I. We can then express $\bar{S}[t]$ in fixpoint form as a function of $\bar{A}$ and the transitions caused by thread t:

$$\begin{array}{l} \bar{S}[t] = \mathrm{lfp}\, G_t(\bar{A}) \text{ where} \\ G_t(\bar{Y})(X) \stackrel{\text{def}}{=} \Pi_t(I) \cup \{ \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X, \sigma \xrightarrow{t} \sigma' \vee \exists t' \neq t, (\sigma, \sigma') \in \bar{Y}[t'] \} \end{array}$$

i.e., to reach a new program state from a known one, we either execute a step from thread t (in $\xrightarrow{t}$) or a step from another thread (in $\bar{A}[t']$). Moreover, we can express directly $\bar{A}$ as a function of $\bar{S}$:

$$\bar{A}[t] = \bar{B}(\bar{S})[t] \text{ where } \bar{B}(\bar{Y})[t] \stackrel{\text{def}}{=} \{ (\sigma, \sigma') \mid \pi_t(\sigma) \in \bar{Y}[t] \wedge \sigma \xrightarrow{t} \sigma' \}$$

which expresses that $\bar{A}[t]$ corresponds to the transitions in τ_t starting from a reachable state. This yields the following *nested fixpoint* characterisation of $\bar{S}$:

$$\bar{S} = \mathrm{lfp}\, \bar{H} \text{ where } \bar{H}(\bar{X})[t] \stackrel{\text{def}}{=} \mathrm{lfp}\, G_t(\bar{B}(\bar{X})) \tag{9}$$

which can be computed in iterative form as: $\bar{S} = \bigcup_{n \in \mathbb{N}} \bar{H}^n(\bar{\emptyset})$ while $\bar{H}(\bar{X})[t] = \bigcup_{n \in \mathbb{N}} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. The benefit of this characterization is that the computation of $\bar{H}(\bar{X})[t]$ only depends on its argument $\bar{X}$ and the transition relation of the thread t, not on the transition relations of the other threads: the inner fixpoint is thus similar to the analysis of a sequential process. However, as analyzing a thread in a given set of interferences $\bar{B}(\bar{X})$ gathers a new, larger set of interferences $\bar{B}(\bar{H}(\bar{X}))$, the analysis must be performed again with this enriched set until it becomes stable. This is the role of the outer fixpoint.

3.5 Memory and Interference Abstractions

Our interference semantic $\bar{S}$ is very concrete (the state semantics $\alpha^{state}(T)$ can be recovered fully from it), and too large to be computed directly. An effective and efficient analyzer can be designed by applying abstractions independently to local invariants and to interferences, while keeping the nested fixpoint form of (9). Firstly, any memory abstraction, such as intervals (4) or polyhedra [10], can be applied directly to each local invariant $\bar{S}[t]$. As the number of variables is often a critical parameter in the efficiency of abstract domains, a faster but coarser analysis can be achieved by removing the auxiliary variables from $\bar{S}[t]$ with α^{ctrl}_t (7) prior to memory abstraction. Secondly, we abstract interferences $\bar{A}[t]$. For instance, it is possible to forget the control state of all threads using a *flow-insensitive* abstraction α^{flow}:

$$\alpha^{flow}(X) \stackrel{\text{def}}{=} \{ (m, m') \mid \exists \bar{\ell}, \bar{\ell}', ((\bar{\ell}, m), (\bar{\ell}', m')) \in X \}$$

Intuitively, this means that, when analyzing a thread t, we consider that any instruction from any thread $t' \neq t$ may be executed at any point of thread t. Then, by noting that $\mathscr{P}((\mathscr{V} \to \mathbb{V}) \times (\mathscr{V} \to \mathbb{V})) \simeq \mathscr{P}((\{1,2\} \times \mathscr{V}) \to \mathbb{V})$, we can see a set of pairs of maps on $\mathscr{V}$ as a set of maps on $\{1,2\} \times \mathscr{V}$, with twice as many variables; hence, we can apply any memory abstraction to $\alpha^{flow}(X)$. Another solution consists in abstracting the relation $\alpha^{flow}(X)$ with its image, using the following abstraction α^{img}, before applying a memory abstraction:

$$\alpha^{img}(X) \stackrel{\text{def}}{=} \{ m' \mid \exists m, (m, m') \in X \}$$

which is coarser but more efficient as it avoids doubling the number of variables. By combining α^{flow}, α^{img}, and α^{cart} (3), we abstract thread interferences in a non-relational and flow-insensitive way, as the set of values each thread can store into each variable during the thread execution. Further abstractions, such as intervals α^{itv} (4), can be applied. This yields an efficient analysis: the abstract interference is simply a global map from $\mathscr{T} \times \mathscr{V}$ to intervals, and we can build an abstraction $\bar{H}^\sharp(\bar{X}^\sharp)[t]$ of $\bar{H}(\bar{X})[t]$ by combining abstractions for each instruction of thread t (as Ex. 5 but slight modified to take interferences into account).

Example 10. The interval abstraction $[\![{}^{\ell}x := y + 1^{\ell'}]\!]^\sharp$ in the environment $[x \mapsto [l_x;h_x]; y \mapsto [l_y;h_y]]$ at ℓ and the global interference map $[x \mapsto [l'_x;h'_x]; y \mapsto [l'_y;h'_y]]$ gives, at ℓ', the interval $x \mapsto [\min(l'_x, l_x + \min(l_y,l'_y)+1); \max(h'_x, h_x + \max(h_y,h'_y)+1)]$. ∎

Example 11. We analyze Fig. 3 using intervals, by applying α^{ctrl}_t, α^{cart}, and α^{itv} to local invariants, and α^{flow}, α^{img}, α^{cart}, and α^{itv} to interferences. We find that, at (3), $\mathtt{x} \in [0;9]$. The information that $\mathtt{x} < \mathtt{y}$ is lost. The second thread produces the abstract interferences $[\mathtt{x} \mapsto \emptyset, \mathtt{y} \mapsto [0;10]]$, i.e., we infer that it cannot modify x and can only put values from $[0;10]$ into y. We lose the information that y can only increase. ∎

Example 12. Consider the program in Fig. 4. When analyzing the first thread, we note that, at (2), x can be 0 or -2, depending on whether the second thread is at (6) or (7). Moreover, to prove that x stays in $\{-2,0\}$ at (2), it is necessary to infer that the interference from the second thread can only increment x when it is at (7), and decrement it when it is at (6). A flow-insensitive abstraction using α^{ctrl}_t and α^{flow} would instead infer that x can be incremented arbitrarily and decremented arbitrarily. It would not allow us to prove that x is bounded. ∎

3.6 Further Considerations

A practical consequence of (9) is that a thread-modular static analyzer can be designed by slightly modifying a sequential analyzer: all we need to do is keep track of (abstract) interferences as well as reachable states, and add an external fixpoint iteration to re-analyze each thread until the inferred interferences stabilize. As for memory abstractions, there are many choices in how to abstract interferences, with various cost versus precision trade-offs and various levels of expressiveness. Although we presented only flow-insensitive non-relational abstract interferences, we can envision abstractions keeping (at least partially) flow and relational information. This is in contrast to our earlier formalization [16] which only allowed flow-insensitive non-relational interferences, and can now be seen a special case of the framework presented here. In the presence of mutual exclusion primitives and scheduling policies that restrict the interleaving of threads (such as thread priorities), a model of the scheduler can be incorporated into the semantics to refine the notion of interference. For instance [16] handles locks by partitioning flow-insensitive, non-relational abstract interferences with respect to the set of locks each thread holds. Another remark is that the interleaved trace semantics (6) is not realistic in the context of modern multicore architectures and compilers; non-coherent caches and optimizations can expose extra behaviors. However, we proved in [16] that, provided that a flow-insensitive abstraction of interferences is used, the abstract semantics is sound even in the presence of weakly consistent memory models, so that the results of the static analysis can still be trusted in realistic settings. Finally, we did not discuss the case of an unbounded number of threads, which requires further abstraction.

4 Applications

We now briefly describe two static analysis tools that were designed at the cole normale suprieure using abstract interpretation principles.

4.1 Astre

Astre is a static analyzer that checks for run-time errors in synchronous (hence sequential) embedded critical C programs. It covers the full C language, including pointers and floats, but excluding dynamic memory allocation and recursivity (forbidden in the realm of critical software). It checks for arithmetic overflows, and invalid arithmetic and memory operations. It is sound, and so, never misses any existing error. Due to over-approximations, it can signal false alarms. Although Astre can analyze many kinds of codes, it is specialized to control-command avionic software, on which it aims at efficiency and certification, i.e., zero false alarm. It was able to prove quickly (2h to 53h) the absence of run-time error in large (100 K to 1M lines) industrial avionic codes. This result could be achieved using a specialization process: starting from a fast and coarse analyzer based on intervals, we added manually more powerful abstract domains as they were needed. Some were borrowed from the large library of existing abstractions, and others were developed specifically for the avionic application domain (such as an abstraction of digital filters [12]). Astre is a mature tool: it is industrialized and made commercially available by AbsInt [1]. More details can be found in [2].

4.2 AstreA

AstreA aims at checking for run-time errors in multithreaded embedded critical C programs. It is based on the Astre code-base and inherits all of its abstractions. Additionally, it implements the interference fixpoint analysis described in Sec. 3. Interferences are abstracted in a flow-insensitive and non-relational way, which was sufficient to give encouraging experimental results. On our target application, a 1.6 Mlines industrial avionic software with 15 threads, AstreA reports around 1 300 alarms in 50h. More details can be found in [16, 2]. AstreA is a research prototype in development; we are currently improving it with more powerful memory and interference abstractions.

5 Conclusion

We have provided in this article a short glimpse of abstract interpretation theory and its application to the systematic construction of static analyzers for sequential and parallel programs. Two key reasons make abstract interpretation-based constructions attractive. Firstly, its ability to relate in a formal way the output of a static analyzer all the way down to the precise dynamic behavior of the input program (its execution traces), both in terms of soundness proof and information loss. Secondly, the ability to decompose the design of an analyzer as the combination of independent, reusable abstractions, which allows the modular implementation of analyzers (e.g., abstract domains designed for Astre could be reused in AstreA). Future work includes the design of new abstractions to improve the AstreA analyzer prototype and reduce the number of alarms on selected embedded multithreaded C programs. In particular, the derivation of thread-modular abstract semantics for parallel programs through the use of concrete interferences we introduced here opens the way to the design of flow-sensitive and relational interference abstractions, which was not possible in the earlier framework underlying AstreA.

Acknowledgments. We thank the anonymous referees on a earlier version of [16] for pointing out the link between the analysis performed by AstreA and rely/guarantee methods, which was not apparent to us before. We also thank Patrick Cousot for urging us to provide a sequence of explicit abstraction functions detailing precisely which information is lost at each step of our construction, instead of providing a monolithic soundness proof as we did in [16].

References

[1] AbsInt, Angewandte Informatik. Astrée run-time error analyzer. `http://www.absint.com/astree`.

[2] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.

[3] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993.

[4] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, Jan. 1977.

[6] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[7] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.

[8] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Procs. of the 2d Int. Conf. on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet (SSGRR'01)*. Scuola Superiore G. Reiss Romoli, Aug. 2001.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, Dec. 2006.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[11] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles*. Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, 15 Nov. 1985.

[12] J. Feret. Static analysis of digital filters. In *Proc. of the 13th Europ. Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 2004.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[14] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. Phd thesis, Oxford University, Jun. 1981.

[15] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.

[16] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):1–63, Mar. 2012.

[17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[19] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proc. of the 9th Int. Symp. on Static Analysis (SAS02)*, volume 2477 of *LNCS*, pages 36–51, 2002.

A Proofs

All the results from Sect. 2 are very classic, so, we do not prove them here (see for instance [4]). Instead, we focus on the new results presented in Sect. 3: the nested fixpoint characterization of the reachable states of a parallel program using interferences.

A.1 Proof of (6)

The proof that the operator F in (6) indeed has a least fixpoint, T, is analogous to that of F in (1). As F is a continuous morphism in the complete partial order of trace sets ordered by inclusion, i.e., $F(\bigcup_i X_i) = \bigcup_i F(X_i)$, we can apply Kleene's fixpoint theorem [4], which states that $\mathrm{lfp}\, F$ exists and is exactly equal to $\bigcup_{n\in\mathbb{N}} F^n(\emptyset)$.

We now repeat this classic proof for the sake of completeness. Let us note $X \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. We first prove that X is indeed a fixpoint of F, then that it is smaller than any other fixpoint. Indeed, $F(X) = F(\bigcup_{n\in\mathbb{N}} F^n(\emptyset)) = \bigcup_{n\in\mathbb{N}} F^{n+1}(\emptyset)$, by continuity. Moreover, $F^0(\emptyset) = \emptyset \subseteq F^1(\emptyset)$, hence $F(X) = \bigcup_{n\in\mathbb{N}} F^{n+1}(\emptyset) \cup F^0(\emptyset) = \bigcup_{n\in\mathbb{N}} F^n(\emptyset) = X$. Hence, X is a fixpoint. Let Y be another fixpoint, i.e., $F(Y) = Y$. We have $F^0(\emptyset) = \emptyset \subseteq Y$ and, if $F^n(\emptyset) \subseteq Y$, then $F^{n+1}(\emptyset) = F(F^n(\emptyset)) \subseteq F(Y)$, as the continuity of F implies its monotonicity, hence $F^{n+1}(\emptyset) \subseteq Y$ as $Y = F(Y)$. Thus, by recurrence, $\forall n, F^n(\emptyset) \subseteq Y$, which implies that $X \subseteq Y$. Hence, X is the least fixpoint of F.

A.2 Proof of (3.4)

By definition (3.4), $\bar{S}[t] \stackrel{\text{def}}{=} \Pi_t(\alpha^{state}(T))$ and $T = \mathrm{lfp}\, F$ with $F(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma_0 \stackrel{t_0}{\to} \cdots \stackrel{t_i}{\to} \sigma_{i+1} \mid \sigma_0 \stackrel{t_0}{\to} \cdots \stackrel{t_{i-1}}{\to} \sigma_i \in X \wedge \sigma_i \stackrel{t_i}{\to} \sigma_{i+1} \,\}$. We wish to prove that $\bar{S}[t] = \mathrm{lfp}\, G_t(\bar{A})$, where $G_t(\bar{A})(X) \stackrel{\text{def}}{=} \Pi_t(I) \cup \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X, \sigma \stackrel{t}{\to} \sigma' \vee \exists t' \neq t, (\sigma,\sigma') \in \bar{A}[t'] \,\}$. To lighten notations, we note $G \stackrel{\text{def}}{=} G_t(\bar{A})$.

We first prove by recurrence on n that $\Pi_t(\alpha^{state}(F^n(\emptyset))) = G^n(\emptyset)$. Intuitively, $G^n(\emptyset)$ corresponds to the states, projected on thread t, that can be reached from I after at most n execution steps. When $n = 0$, $\Pi_t(\alpha^{state}(F^0(\emptyset))) = G^0(\emptyset) = \emptyset$. Assume that the property holds for some n and let us prove that it holds for $n+1$. As π_t is one-to-one, we can reason equivalently on $G^{n+1}(\emptyset)$ and the states σ such that $\pi_t(\sigma) \in G^{n+1}(\emptyset)$, so, let us consider such a σ. Then, either (a) $\sigma \in I$, or σ can be reached from some σ' such that $\pi_t(\sigma') \in G^n(\emptyset)$ and either (b) $\sigma' \stackrel{t}{\to} \sigma$ or (c) $\exists t' \neq t, (\sigma',\sigma) \in \bar{A}[t']$. By definition, $(\sigma',\sigma) \in \bar{A}[t']$ is equivalent to the existence of a trace $\cdots\sigma' \stackrel{t'}{\to} \sigma\cdots$ in T. Moreover, by recurrence hypothesis, $\sigma' \in \alpha^{state}(F^n(\emptyset))$, i.e., there exists a trace in T with at most n transitions and ending in σ'. This means that, if a transition $\cdots\sigma' \stackrel{t'}{\to} \sigma\cdots$ exists in a trace in T, it also exists in a trace of length at most $n+1$ in T (the set of traces is "closed by fusion" [11]). Hence, case (c) is equivalent to the existence of $t' \neq t$ and a trace $\cdots\sigma' \stackrel{t'}{\to} \sigma\cdots$ in $F^{n+1}(\emptyset)$. Case (b) is equivalent to the existence of a trace

$\cdots\sigma' \xrightarrow{t} \sigma\cdots$ in $F^{n+1}(\emptyset)$. Thus, the disjunction of (a), (b), and (c) is equivalent to the existence of a trace in $F^{n+1}(\emptyset)$ containing σ, which is equivalent to $\sigma \in \alpha^{state}(F^{n+1}(\emptyset))$. This ends the proof by recurrence.

We now use the characterization of fixpoints by iteration: we proved in the proof of (6) that $T = \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. Kleene's theorem implies likewise that $\text{lfp}\, G = \bigcup_{n\in\mathbb{N}} G^n(\emptyset)$. By continuity of Π_t and α^{state}, we get that $\Pi_t(\alpha^{state}(T)) = \Pi_t(\alpha^{state}(\bigcup_{n\in\mathbb{N}} F^n(\emptyset))) = \bigcup_{n\in\mathbb{N}} \Pi_t(\alpha^{state}(F^n(\emptyset)))$. Applying the result of the preceding paragraph, we get $\Pi_t(\alpha^{state}(T)) = \bigcup_{n\in\mathbb{N}} G^n(\emptyset) = \text{lfp}\, G_t(\bar{A})$.

A.3 Proof of (3.4)

We now prove that $\bar{A}[t] = \{\,(\sigma,\sigma') \mid \pi_t(\sigma) \in \bar{S}[t] \wedge \sigma \xrightarrow{t} \sigma'\,\}$. By definition, $\bar{A}[t] \overset{\text{def}}{=} \{\,(\sigma_i,\sigma_{i+1}) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T, t_i = t\,\}$. As T is closed by prefix, we have $\bar{A}[t] = \{\,(\sigma_{n-1},\sigma_n) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T, t_{n-1} = t\,\}$. As T is a fixpoint of F, $\exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T$ is equivalent to $\exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-2}} \sigma_{n-1} \in T$ and $\sigma_{n-1} \xrightarrow{t_{n-1}} \sigma_n$. Finally, as $\bar{S}[t] = \Pi_t(\alpha^{state}(T))$, this is equivalent to $\pi_t(\sigma_{n-1}) \in \bar{S}[t]$ and $\sigma_{n-1} \xrightarrow{t_{n-1}} \sigma_n$, which ends the proof.

A.4 Proof of (9)

We now prove that $\bar{S} = \text{lfp}\, \bar{H}$, where $\bar{H}(\bar{X})[t] \overset{\text{def}}{=} \text{lfp}\, G_t(\bar{B}(\bar{X}))$, and give the iterative form of $\text{lfp}\, \bar{H}$.

Firstly, we prove that $\bar{S}$ is a fixpoint of $\bar{H}$. Indeed, for any t, $\bar{H}(\bar{S})[t] = \text{lfp}\, G_t(B(\bar{S})) = \text{lfp}\, G_t(\bar{A})$ by (3.4), which equals $\bar{S}[t]$ by (3.4).

Secondly, we prove that $\text{lfp}\, \bar{H}$ can be expressed in iterative form. To do so, we prove that $\bar{H}$ is continuous. Kleene's fixpoint theorem applied to the continuous function G_t gives the following characterization of $\bar{H}$: $\bar{H}(\bar{X})[t] = \bigcup_{n\in\mathbb{N}} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. To simplify notations, we define, for each t and n, the function $\bar{I}_n(\bar{X})[t] \overset{\text{def}}{=} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. Then, we have $\bar{H}(\bar{X}) = \bar{\bigcup}_{n\in\mathbb{N}} \bar{I}_n(\bar{X})$. We note that each $\bar{I}_n$ is continuous. Hence, we have: $\bar{H}(\bar{\bigcup}_i \bar{X}_i) = \bar{\bigcup}_{n\in\mathbb{N}} \bar{I}_n(\bar{\bigcup}_i \bar{X}_i) = \bar{\bigcup}_{n\in\mathbb{N}} \bar{\bigcup}_i \bar{I}_n(\bar{X}_i) = \bar{\bigcup}_i \bar{\bigcup}_{n\in\mathbb{N}} \bar{I}_n(\bar{X}_i) = \bar{\bigcup}_i \bar{H}(\bar{X}_i)$, which proves the continuity of $\bar{H}$. We can thus apply Keene's fixpoint theorem to $\bar{H}$ itself to get $\text{lfp}\, \bar{H}$ in iterative form: $\text{lfp}\, \bar{H} = \bar{\bigcup}_{n\in\mathbb{N}} \bar{H}^n(\bar{\emptyset})$.

Finally, we prove that $\bar{S} \bar{\subseteq} \text{lfp}\, \bar{H}$. To do this, we prove by recurrence on n that, if σ_n is reachable after n steps, i.e., if there exists some trace $\sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n$ in T, then, for any t, $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$. When $n = 0$, $\bar{H}(\bar{\emptyset})[t] = \text{lfp}\, G_t(\bar{B}(\emptyset)) = \text{lfp}\, G_t(\emptyset) \supseteq \Pi_t(I)$. As $\sigma_0 \in I$, we indeed have $\pi_t(\sigma_0) \in \Pi_t(I)$. Assume that the property is true at rank n and consider a trace $\sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_n} \sigma_{n+1}$ in T. As T is closed by prefix, $\sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n$ is also in T and we can apply the recurrence hypothesis to get $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$. Moreover, we have $\sigma_n \xrightarrow{t_{n+1}} \sigma_{n+1}$. We consider first the case $t_{n+1} = t$. Then as $(\bar{H}^{n+1}(\bar{\emptyset}))[t]$ is closed by reachability from thread t, we also have $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$, and so, $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+2}(\bar{\emptyset}))[t]$. Consider now the case $t_{n+1} \neq t$. Then, $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$ implies that $(\sigma_n,\sigma_{n+1}) \in \bar{B}(\bar{H}^{n+1}(\bar{\emptyset}))[t]$. As a consequence, $(\bar{H}^{n+2}(\bar{\emptyset}))[t]$ is closed under reachability through the transition $\sigma_n \xrightarrow{t_{n+1}} \sigma_{n+1}$, and so, $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+2}(\bar{\emptyset}))[t]$. We have proved that, for any $\pi_t(\sigma) \in \bar{S}[t]$, there exists some n such that $\pi_t(\sigma) \in (\bar{H}^n(\bar{\emptyset}))[t]$. As $\text{lfp}\, \bar{H} = \bar{\bigcup}_{n\in\mathbb{N}} \bar{H}^n(\bar{\emptyset})$, we get that $\pi_t(\sigma) \in (\text{lfp}\, \bar{H})[t]$, hence, $\bar{S} \bar{\subseteq} \text{lfp}\, \bar{H}$.

Verification of concurrent systems

Ahmed Bouajjani

LIAFA, Paris Diderot University & CNRS, France

The verification of concurrent programs is a challenging problem. This is due to the huge number of orderings in which actions of different threads can be executed, and to the intricacy of the interactions between these threads (especially in presence of dynamic thread creation, recursion, etc). Basic problems such as the state reachability problem, that is relevant in checking safety properties, are undecidable in general, even when the manipulated data are in a finite domain. Therefore, restrictions either on the considered class of program models or on the class of explored behaviors during the analysis, must be considered in order to obtain decidable and/or tractable analysis problems. In this talk, we will present program models capturing relevant classes of programs (including for instance asynchronous programs) and study the decidability and complexity of their state reachability problem. Moreover, we will present bounded analysis techniques (such as context-bounding) that are used for efficient bug detection in concurrent programs.

Software Synthesis

Ruzica Piskac

Max Planck Institute for Software Systems
Germany

piskac@mpi-sws.org

Software synthesis is a technique for automatically generating code given a specification. The goal of software synthesis is to make coding easier while increasing both the productivity of the programmer and the correctness of the produced code. Code produced this way is correct by construction.

The idea of automatically generating code was introduced more than 30 years ago [1]. However, it is not always easy to automatically construct code - due to the high computational cost of synthesizing code, this idea was not further explored until recently. There is a regaining interest in software synthesis that is driven by the increasing computational power. Today even desktop machines are able to construct code from complex input specifications. On the other hand, it is not only the computation power that plays an important role in synthesizing code. Recently we have witnessed to a rapid progress of automated reasoning. There are various tools that can automatically prove formulas belonging to different logics. Using automated reasoning we can handle complex specifications by employing efficient algorithms for reasoning about the domain of the specification.

In this lecture we will present our new approach to synthesis that relies on the use of automated reasoning and decision procedures [2, 4, 5]. We handle complex specifications by employing efficient algorithms for reasoning about the domain of the specification. We will describe how to generalize decision procedures into predictable and complete synthesis procedures. Here completeness means that the procedure is guaranteed to find code that satisfies the given specification. Moreover, the synthesis procedure also outputs preconditions on input values that guarantee the existence of the output values. As an example, I will explain how to transform a decision procedure for linear arithmetic into such a synthesis procedure.

In addition, we will also outline a synthesis procedure for specifications given in the form of type constraints [3]. The procedure takes into account polymorphic type constraints as well as code behavior and derives code snippets that use given library functions. We use a first-order resolution-based theorem prover to solve these constraints and derive code snippets. The constraints can have multiple solutions and hence the theorem prover may produce more than one code snippet. Therefore, we use an additional weight function to rank the derived snippets.

References

[1] Zohar Manna, Richard J. Waldinger. A Deductive Approach to Program Synthesis. In *ACM Trans. Program. Lang. Syst.*, 2(1): 90-121 (1980).

[2] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter. Software Synthesis Procedures. In *Commun. ACM*, 55(2): 103-111 (2012).

[3] Tihomir Gvero, Viktor Kuncak, Ruzica Piskac. Interactive Synthesis of Code Snippets. In *CAV 2011*, 418-423.

[4] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter. Comfusy: A Tool for Complete Functional Synthesis. In *CAV 2010*, 430-433.

[5] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter. Complete Functional Synthesis. In *PLDI 2010*, 316-329.

Student Papers

Checking Two Structural Properties of Vector Addition Systems with States

Florent Avellaneda
LIF, AMU & CNRS
Marseille, France
florent.avellaneda@lif.univ-mrs.fr

Rémi Morin
LIF, AMU & CNRS
Marseille, France
remi.morin@lif.univ-mrs.fr

Abstract

A p-dimensional vector addition system with states (VASS) is a weighted digraph whose arcs carry a p-dimensional integral vector. The structural boundedness problem asks whether a given VASS admits a closed path with a positive weight vector. Although a VASS can be simulated by a Petri net, none of the known simulations preserve structural boundedness. Thus we cannot cite the classical result by Memmi and Roucairol to assert that checking structural boundedness is polynomial. Based on some encoding in the form of a system of exponentially many linear Diophantine inequalities, we show that checking the structural boundedness of a given VASS is polynomial with the help of a separation algorithm. Similar techniques can be used for checking the structural termination of a VASS, which asks whether there exists some closed path with a non-negative weight vector.

1 Introduction

Consider a set of reactions that take place among a collection of particles such that each reaction consumes a multiset of available particles and produces a linear combination of other particle types. This kind of framework can be formalized by a vector addition system [5] or, equivalently, a (pure) Petri net [8]. Consider in addition some control state which determines whether a reaction can occur or not, and such that the occurrence of a reaction leads to a possibly distinct control state. Then the model becomes formally a vector addition system with states (a VASS), a notion introduced in [4]. In this paper we investigate the computational complexity of two *structural properties* for VASS, that is, properties that do not depend on a particular initial collection of particles. In this way, we consider the initial configuration as a parameter of the system.

The first problem we consider asks whether the number of particles remains bounded, that is, only finitely many distinct configurations can be reached, for each initial configuration. Since particles often represent the consumption of resources, such as messages in channels, this first problem asks whether there exists some amount of resources sufficient to cope with all configurations reachable from any fixed finite set of potential configurations. A second basic issue is to check that a given system terminates, i.e. whether there is no infinite execution, for each initial configuration. In other words, we aim at checking that a system eventually deadlocks. Although one usually tries to avoid deadlocks in concurrent systems, termination remains in some cases a basic problem in formal verification: In particular non-termination can result from livelocks in concurrent programs when components fail to achieve their tasks.

Petri nets form a popular and intensively studied model of distributed systems. Formally a Petri net is simply a VASS with a single control state. It is often claimed that Petri nets, vector addition systems, and vector addition systems with states are equivalent formalisms because they can simulate each other

(see e.g. [8]). The idea is simple: A VASS with n control states can be simulated by a VASS with a single control state (i.e. a Petri net) which makes use of n additional particle types. These n additional particle types are used to encode the current control state of the former VASS by means of a *single* additional particle that evolves from one of these types to another. Since only one particle is allowed within these additional types, *this simulation does not preserve structural properties*. More precisely it is easy to design a structurally bounded VASS whose equivalent Petri net is not structurally bounded.

In order to describe a distributed system, it is often convenient to use a *vector* of control states whose components are the local states of each process. Then particles represent messages within channels. This model is called a *parallel-composition-VASS* [6]. It is close to the notion of a communicating finite state machine but with non-FIFO message exchanges. Similarly to the simulation of a VASS by a Petri net, a parallel-composition-VASS can be simulated by a Petri net with additional places whose marking represent the current vector of control states. Again, structural properties are not preserved by this simulation. Besides checking the structural boundedness of a parallel-composition-VASS is NP-complete [6] whereas this problem can be solved in polynomial time in the particular case of Petri nets [2, 7]. In this paper, we prove that *the problems of checking the structural boundedness and the structural termination of a given VASS can be solved in polynomial time*.

2 Background

Let p be a fixed non-zero natural number. A vector addition system with states is simply a directed graph whose arcs are labeled by vectors from $\mathbb{Z}^p$.

Definition 1. [4] A *vector addition system with states* (for short: a VASS) is a pair $\mathcal{S} = (Q,A)$ where Q is a finite set of states, and $A \subseteq Q \times \mathbb{Z}^p \times Q$ is a finite set of arcs labeled by vectors from $\mathbb{Z}^p$.

Throughout the paper we let $\mathcal{S} = (Q,A)$ be a VASS. We let $|Q|$ and $|A|$ denote the cardinalities of Q and A respectively. The source and the target of a labeled arc $a \in A$ are denoted by $\mathrm{dom}(a)$ and $\mathrm{cod}(a)$ respectively. We let $\mathrm{cost}(a) \in \mathbb{Z}^p$ denote the vector labeling each arc $a \in A$.

2.1 Basics and notations

Let $\mathcal{S} = (Q,A)$ be a VASS. A path is a sequence of arcs $\gamma = a_1...a_n \in A^\star$ such that we have $\mathrm{dom}(a_{i+1}) = \mathrm{cod}(a_i)$ for each $i \in [1..n-1]$. A path $\gamma = a_1...a_n \in A^\star$ is *closed* if $n \geq 1$ and $\mathrm{dom}(a_1) = \mathrm{cod}(a_n)$. A path $\gamma = a_1...a_n \in A^\star$ is *simple* if $\mathrm{dom}(a_i) \neq \mathrm{dom}(a_j)$ for all distinct i, j. A *circuit* is a simple and closed path. The cost of a path $\gamma = a_1...a_n$ is the vector $\mathrm{cost}(\gamma) = \sum_{i=1}^{i=n} \mathrm{cost}(a_i)$. Further the cost of a multiset of arcs $x \in \mathbb{N}^A$ is $\mathrm{cost}(x) = \sum_{a \in A} x(a) \cdot \mathrm{cost}(a)$ and the cost of a finite multiset of paths $\mathscr{F}$ is $\mathrm{cost}(\mathscr{F}) = \sum_{\gamma \in A^\star} \mathscr{F}(\gamma) \cdot \mathrm{cost}(\gamma)$.

A configuration is a pair $(q,r) \in Q \times \mathbb{N}^p$ consisting of a control state q and a multiset of available particles r. A labeled arc $a \in A$ is enabled at the configuration (q,r) and leads to the configuration (q',r') if $\mathrm{dom}(a) = q$, $\mathrm{cod}(a) = q'$, and $r + \mathrm{cost}(a) = r'$. An *execution* of $\mathcal{S}$ from an initial configuration $(q_{\mathrm{in}}, r_{\mathrm{in}})$ is a sequence of labeled arcs $a_1...a_n \in A^\star$ such that there are configurations (q_0,r_0), ..., (q_n,r_n) for which $(q_0,r_0) = (q_{\mathrm{in}}, r_{\mathrm{in}})$ and for each $i \in [1..n]$, the labeled arc a_i is enabled at (q_{i-1}, r_{i-1}) and leads to (q_i,r_i). Then the configuration (q_n,r_n) is *reachable* from $(q_{\mathrm{in}}, r_{\mathrm{in}})$. The set of all executions of $\mathcal{S}$ from an initial configuration $(q_{\mathrm{in}}, r_{\mathrm{in}})$ provided with the prefix partial order over $A^\star$ is called the *reachability tree* of $\mathcal{S}$ from $(q_{\mathrm{in}}, r_{\mathrm{in}})$.

Let x and y be two integral vectors with n components: $x = (x[1],...,x[n])$ and $y = (y[1],...,y[n])$. We put as usual $x \geq y$ if $x[i] \geq y[i]$ for each i; $x > y$ if $x[i] > y[i]$ for each i; and $x \gneq y$ if $x \geq y$ and $x \neq y$. As

explained below, we shall represent closed paths of a VASS $\mathcal{S}$ as particular multisets of labeled arcs.

2.2 Structural properties and characterizations with closed paths

Definition 2. A VASS $\mathcal{S}$ is *structurally terminating* if it is terminating for any initial configuration.

The structural termination problem for vector addition systems asks whether a given VASS has an infinite execution for some initial configuration. We observe first that this question boils down to search for particular closed paths in $\mathcal{S}$.

Proposition 3. *A VASS $\mathcal{S}$ is structurally terminating if and only if there exists no closed path γ with $cost(\gamma) \geq \vec{0}$.*

A VASS provided with an initial configuration $(q_{\text{in}}, r_{\text{in}})$ is *bounded*, if there is a natural number k such that, for each configuration (q,r) reachable from $(q_{\text{in}}, r_{\text{in}})$, there are never more than k particles of each type, i.e. $r[i] \leq k$ for each $i \in [1..p]$.

Definition 4. A VASS $\mathcal{S}$ is *structurally bounded* if it is bounded for all initial configurations.

Similarly to Proposition 3, checking the structural boundedness of a VASS boils down to detect a closed path with a non-negative non-zero cost.

Proposition 5. *A VASS $\mathcal{S}$ is structurally bounded if and only if there exists no closed path γ with $cost(\gamma) \gneq \vec{0}$.*

2.3 Connected Eulerian multisets of labeled arcs vs. closed paths

Definition 6. A multiset of labeled arcs $x \in \mathbb{N}^A$ is called *Eulerian* if for each state $q \in Q$ we have $\sum_{\text{dom}(a)=q} x(a) = \sum_{\text{cod}(a)=q} x(a)$.

Proposition 7. *(Euler's theorem) Let $x \in \mathbb{N}^A$ be a non-empty connected and Eulerian multiset of labeled arcs. Then there exists some closed path $\gamma = a_1 \ldots a_n \in A^\star$ such that $\sum_{i=1}^{i=n} a_i = x$.*

It is easy to show that structural boundedness and structural termination are in NP: A witness of a pathological path is simply a connected subset of arcs $C \subseteq A$ for which there exists an Eulerian set of arcs x such that $A_x = C$ and $\text{cost}(x) \gneq 0$ (resp. $\text{cost}(x) \geq 0$).

3 From pathological closed paths to weak-circuits

In this section we present the key technical lemma of this work (Lemma 9). This result makes use of the notion of weak-circuits. Each pathological path for structural termination (or structural boundedness) with inner state $\hat{q}$ can be represented by a multiset of weak-circuits starting from the base state $\hat{q}$. Moreover we prove that the length of these weak-circuits can be bounded.

3.1 Definition and examples

In this section we fix a VASS $\mathcal{S} = (Q,A)$ and a base state $\hat{q} \in Q$.

Definition 8. Let $q \in Q$ be a state of $\mathcal{S}$ and γ_0 be a circuit of $\mathcal{S}$ starting from q. Let γ_1 be a simple path from $\hat{q}$ to q and γ_2 be a simple path from q to $\hat{q}$. Let $k \in \mathbb{N}$. If the length of the path $\gamma_1.(\gamma_0)^k.\gamma_2$ from $\hat{q}$ to $\hat{q}$ is at least 1, then the closed path $\gamma_1.(\gamma_0)^k.\gamma_2$ is called a *weak-circuit* of $\mathcal{S}$ with *valuation* k starting from $\hat{q}$.

A weak-circuit will be often represented by a multiset of arcs $W = D + k \cdot C$ where D is the multiset of arcs occurring in γ_1 and γ_2 and C is the set of arcs occurring in γ_0. Then the multiset W is connected and Eulerian. Note that the path $\gamma_1.\gamma_2$ from q to q need not to be simple (nor non-empty). However each arc occurs at most twice in $\gamma_1.\gamma_2$.

3.2 From closed paths to weak-circuits with a bounded valuation

We are now ready to prove the expected Lemma 9.

Lemma 9. *If $\hat{q}$ is an inner state of a closed path γ such that $cost(\gamma) \geq \vec{0}$ then there exists a non-empty finite family of weak-circuits W_1, ..., W_n starting from $\hat{q}$ with valuation at most 2^{Φ} where $\Phi = 96 \times (p+1)^4 \times size(\mathcal{S})$ and some non-zero natural numbers λ_1,..., $\lambda_n \in \mathbb{N}^*$ such that $\sum_{i=1}^{i=n} \lambda_i \cdot cost(W_i) \geq \vec{0}$.*

4 Checking structural termination

In this section we explain why checking the structural termination of a given VASS can be done in polynomial time. Let $\mathcal{S}$ be a VASS and $\hat{q} \in Q$ be a fixed state of $\mathcal{S}$. We build a system of linear Diophantine inequalities $S_{\mathcal{S},\hat{q}}$ which has a solution if and only if there exists some closed path γ with inner state $\hat{q}$ such that $\text{cost}(\gamma) \geq \vec{0}$. Then Prop. 3 guaranties that the VASS $\mathcal{S}$ terminates structurally if and only if the system $S_{\mathcal{S},\hat{q}}$ has no solution for each $\hat{q} \in Q$. The system $S_{\mathcal{S},\hat{q}}$ derives from Lemma 9. Observe first that the number N of weak circuits starting from $\hat{q}$ with valuation at most 2^{Φ} is exponential in the size of $\mathcal{S}$. It is easy to design a system of p linear Diophantine inequalities with N unknown natural numbers which has a solution if and only if there is a finite family of weak-circuits starting from $\hat{q}$ with valuation at most 2^{Φ} whose cost is non-negative.

We consider actually the dual problem. The system $S_{\mathcal{S},\hat{q}}$ relies on a vector w of p unknown components such that $w > \vec{0}$. It consists essentially of the inequality $\text{cost}(\gamma)^{\top} w < 0$ for each weak-circuit γ starting from $\hat{q}$ with valuation at most 2^{Φ}. With the help of the Infeasibility Theorem [1, Th. 2.1], the system $S_{\mathcal{S},\hat{q}}$ has no solution if and only if there exists some non-negative non-zero linear combination of such weak-circuits whose cost is non-negative. By Lemma 9, this is equivalent to the existence of a closed path γ with inner state $\hat{q}$ such that $\text{cost}(\gamma) \geq \vec{0}$. However, $S_{\mathcal{S},\hat{q}}$ consists of possibly exponentially many inequalities. For that reason, we design a separation oracle to solve this system in polynomial time with the help of the fundamental result due to Grötschel, Lovász and Schrijver [3] (see also [9, Th. 14.1]).

Given a vector $w > \vec{0}$, the separation oracle decides whether w is a solution to $S_{\mathcal{S},\hat{q}}$ or not, and, in the latter case, it must compute an inequality of $S_{\mathcal{S},\hat{q}}$ for which w fails. At this point the separation oracle need to compute a particular weak-circuit. However, computing such a weak-circuit turns out to be NP-hard in general. In order to cope with this difficulty, we introduce below the notion pseudo-circuits, which are particular closed paths. Since each weak-circuit is a pseudo-circuit, we may replace weak-circuits by pseudo-circuits in the statement of Lemma 9. Consequently, the system $S_{\mathcal{S},\hat{q}}$ we consider consists actually of the inequation $\text{cost}(\gamma)^{\top} w < 0$ for each pseudo-circuit γ with valuation at most 2^{Φ}. In this way, the separation oracle compute a pseudo-circuit γ for which $\text{cost}(\gamma)^{\top} \cdot w > 0$ whenever w is not a solution to $S_{\mathcal{S},\hat{q}}$.

4.1 Searching for bounded pseudo-circuits with inequations

The system of inequalities we consider relies on the following notion of pseudo-circuits which generalizes the definition of a weak-circuit.

Algorithm 1 (Separation algorithm)

Require: $\mathcal{S} = (Q,A)$ is a VASS, $w \in \mathbb{Q}^p$, $\hat{q} \in Q$.
Ensure: returns `true` if w is a solution to $S_{\mathcal{S},\hat{q}}$ and some violated inequality otherwise

```
if w ≯ 0⃗ then
  return some i ∈ [1..p] such that w[i] ≤ 0.
end if
for q, q' ∈ Q do
  Compute blmw_{q,q'}(w) ∈ ℚ and some γ_{q,q'} ∈ A* in polynomial time
end for
for q ∈ Q do
  if (*) blmw_{q̂,q}(w) + 2^Φ × blmw_{q,q}(w) + blmw_{q,q̂}(w) ≥ 0 then
    return cost(γ_{q̂.q}) + 2^Φ × cost(γ_{q,q}) + cost(γ_{q,q̂})
  end if
end for
return true
```

Definition 10. Let $q,q' \in Q$ be two states of $\mathcal{S}$. Let γ_0 be a closed path of $\mathcal{S}$ starting from q'. Let γ_1 be a path from q to q' and γ_2 be a path from q' to q. Let $k \in \mathbb{N}$. We assume that the lengths of γ_0, γ_1 and γ_2 are at most $|Q|$. Let $W = \gamma_1.\gamma_0^k.\gamma_2$ be the closed path which starts from q and which consists of γ_1, followed by k iterations of the circuit γ_0, followed by γ_2. If the length of W is at least 1 then W is called a *pseudo-circuit* of $\mathcal{S}$ with *valuation* k.

Noteworthy each weak-circuit with valuation k (Def. 8) is a pseudo-circuit with valuation k. Consequently Lemma 9 still holds if one replace weak-circuits by pseudo-circuits.

We let $S_{\mathcal{S},\hat{q}}$ be the system of linear inequalities for $w \in \mathbb{Q}^p$ consisting of the following constraints:

- for each $i \in [1..p]$: $w[i] > 0$
- for each pseudo-circuit W starting from $\hat{q}$ with valuation $\leq 2^\Phi$: $-\text{cost}(W)^\top w > 0$.

Proposition 11. *The system $S_{\mathcal{S},\hat{q}}$ has no solution if and only if there exists some closed path γ with inner state $\hat{q}$ such that cost$(\gamma) \geq \vec{0}$.*

4.2 Separation of solutions

Let $w \in \mathbb{Q}^p$. If some component $w[i]$ of w is non-positive, then the constraint $w[i] > 0$ is not satisfied. Thus we may assume that $w > \vec{0}$. We denote by $\mathcal{S}/w = (Q,A/w)$ the directed graph obtained from the VASS $\mathcal{S}$ by replacing the label $\text{cost}(a) \in \mathbb{Z}^p$ of each arc $a \in A$ by $\text{cost}(a)^\top w$. For any two states $q,q' \in Q$, we compute the maximal weight $\text{blmw}_{q,q'}(w) \in \mathbb{Q}$ of the paths from q to q' in $\mathcal{S}/w$ with length at most $|Q|$. We compute also such a maximal path $\gamma_{q,q'} \in A^\star$ with length at most $|Q|$ from q to q', regarded as a path in $\mathcal{S}$. This can be done, e.g., by means of $|Q|$ matrix multiplications in $(\max,+)$-algebra. Note that $\text{blmw}_{q,q}(w) \geq 0$ for each $q \in Q$. Let $q \in Q$ be some state of $\mathcal{S}$. If $\text{blmw}_{\hat{q},q}(w) + 2^\Phi \times \text{blmw}_{q,q}(w) + \text{blmw}_{q,\hat{q}}(w) \geq 0$ then the pseudo-circuit W built from the path $\gamma_{\hat{q},q}$, followed by 2^Φ iterations of the closed path $\gamma_{q,q}$ and the path $\gamma_{q,\hat{q}}$ satisfies $\text{cost}(W)^\top w \geq 0$ because $\text{cost}(W) = \text{cost}(\gamma_{\hat{q},q}) + 2^\Phi \times \text{cost}(\gamma_{q,q}) + \text{cost}(\gamma_{q,\hat{q}})$. This leads us to Algorithm 1.

Proposition 12. *Algorithm 1 decides whether w is a solution to $S_{\mathcal{S},\hat{q}}$ or not, and, in the latter case, returns an inequality of $S_{\mathcal{S},\hat{q}}$ for which w fails.*

As a consequence, we can apply [9, Th. 14.1] and get

Theorem 13. *We can check the structural termination of a given VASS in polynomial time.*

4.3 Checking structural boundedness

We show now how to check whether a given VASS $\mathcal{S}$ is structurally bounded. We let $S^{\circ}_{\mathcal{S},\hat{q}}$ be the system of linear inequalities for $w \in \mathbb{Q}^p$ consisting of the following constraints:

- for each $i \in [1..p]$: $w[i] > 0$
- for all pseudo-circuits W starting from $\hat{q}$ with valuation $\leq 2^{\Phi}$: $\text{cost}(W)^{\top} w \leq 0$.

We can adapt the proof of Prop. 11 and show that $S^{\circ}_{\mathcal{S},\hat{q}}$ characterizes the structural boundedness of $\mathcal{S}$.

Proposition 14. *The system $S^{\circ}_{\mathcal{S},\hat{q}}$ has no solution if and only if there exists some closed path γ with inner state $\hat{q}$ such that $\text{cost}(\gamma) \gneq \vec{0}$.*

It is easy to design a new separation algorithm for $S^{\circ}_{\mathcal{S},\hat{q}}$: we need simply to replace the test $(*)$ $\text{blmw}_{\hat{q},q}(w) + 2^{\Phi} \times \text{blmw}_{q,q}(w) + \text{blmw}_{q,\hat{q}}(w) \geq 0$ from Algorithm 1 by the following condition: $\text{blmw}_{\hat{q},q}(w) + 2^{\Phi} \times \text{blmw}_{q,q}(w) + \text{blmw}_{q,\hat{q}}(w) > 0$. We can prove that the resulting algorithm decides whether w is a solution to $S^{\circ}_{\mathcal{S},\hat{q}}$ or not, and, in the latter case, returns an inequality of $S^{\circ}_{\mathcal{S},\hat{q}}$ for which w fails. Thus,

Theorem 15. *We can check the structural boundedness of a given VASS in polynomial time.*

5 Conclusion

Despite the well-known fact that vector addition systems with states are equivalent to Petri nets when they are provided with an initial configuration, checking the structural properties of a VASS turns out to be much more complicated than for Petri nets. Yet we present polynomial time algorithms to check the structural termination and the structural boundedness of a given VASS by means of an encoding in linear programming and a separation algorithm.

References

[1] G.B. Dantzig and M.N. Thapa. *Linear Programming*. Springer, 2003.

[2] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[3] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[4] J.E. Hopcroft and J-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8:135–159, 1979.

[5] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[6] S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for the boundedness of UML RT models. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2004.

[7] G. Memmi and G. Roucairol. Linear algebra in net theory. In Wilfried Brauer, editor, *Advanced Course: Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 213–223. Springer, 1980.

[8] C. Reutenauer. *The Mathematics of Petri Nets.* New York, USA: Prentice-Hall, 1990.

[9] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

Kaira: HPC and Petri nets

Stanislav Böhm, Ondřej Meca, Martin Šurkovský

Abstract

This paper briefly describes the main aspects and the current state of the tool Kaira. Kaira is a programming environment for creating applications in the area of high-performance computing. The tool uses attributes that are usually seen in modelling tools like simulation or high level approach and provide them as a development environment. Parallel aspects and communication of designed applications are expressed by visual models based on Coloured Petri nets.

1 Introduction

There is no standard definition of high-performance computing (HPC), but it usually means the use of parallel computers for run technical and scientific computations demanding huge numbers of operations. Physics simulations or weather forecasting can be named as examples. Parallel computers provide performance for such tasks but development of applications for these systems is more complex than development of sequential applications. There are many tools and libraries developed with the aim to create these applications easier and more portable[5]. OpenMP[1] and MPI[2][10] are among the most widely used tools in this area. OpenMP is very effective if the core of computation can be expressed as a cycle with independent runs. In such cases programmers are able to get parallel applications from sequential versions quickly. But this effectively runs only on shared memory architectures. MPI is designed for usage with distributed memory, but it is a quite low level interface from an application programmer's point of view. There are also libraries for specific areas besides universal tools. Let us name PetsC[3] and Trilinos[4] as examples for numerical computations. They usually offer typical data structures and parallel operations on them. When we want to use standard algorithms and there is such a library then it is often the best way.

The tool described in this paper is based on generating code from Coloured Petri nets (CPNs)[7]. Usually in this area, somehow restricted CPNs and high-level languages are used [8, 9]. It leads to generation of human-readable codes. We do not introduce such restrictions because we are focused on difference aspects of a code generation. For example, for us it is important to avoid unnecessary memory movements because processed data can be large. This is one of reasons why we use C++ instead of high-level languages.

In this text we introduce a development environment for programming parallel applications, the tool named Kaira. More precisely, Kaira is focused on the area of HPC systems with distributed memory architectures. We want to create a practical tool but on a higher level of abstraction than standard tools. The basic aspects and goals are the follows:

[1] `http://openmp.org/`
[2] `http://www.mpi-forum.org/docs/docs.html`
[3] `http://www.mcs.anl.gov/petsc/`
[4] `http://trilinos.sandia.gov/`

- *Abstract computational model* – The tool should be at the basic level usable by non-experts in parallel/distributed programming. The model should be sufficiently abstract hence it can be used without knowledge of low-level details of used technologies and without solving low level issues. On the other hand our goal is *not* to develop an automatic parallelization tool. Therefore Kaira does not discover parallelisms in applications. The user has to explicitly define them, but they are defined in a high level way and the tool derives implementation details. We also want to use an abstract model to communicate in the opposite way, i.e. from a program to a user during debugging and profiling. We want to show what happens inside a developed program in terms of our high level model without using specialized low level tools.
- *Integration with existing source codes* – Our goal is to create a practical tool, therefore one of the important aspects is reusing of existing codes. We want to achieve integration in both ways. One way is an easy integration of existing libraries in our abstract model and the second way is to integrate resulting code from Kaira into other programs.
- *Fast prototyping* – It can take a long time to get a working prototype during the development of an application for distributed memory systems. We want to allow observing behaviour of an incomplete application from a very early stage of a development cycle. Together with the previous point, we want to allow a smooth iterative development from a sequential version (for example written in C++) to a parallel version (with support of Kaira).
- *Reasonable fast resulting programs* – We are developing a high level tool, but we are still in the area of HPC, therefore performance matters. We can only accept a very small performance loss in compare to hand-optimized solutions.

The result of our efforts is the tool called Kaira[5][1, 4, 2, 3]. Kaira is an open source project released under GPL licence[6].

We have chosen semantics based on CPNs as a computation model for the specification of parallel aspects and communication in applications. CPNs naturally capture parallel behaviour and they also provide natural visual representations of models, visual editing of models and their simulations. However, we do not want to visually program a complete application, our visual language is focused on parallelisms and communication. Sequential parts of a program can be created in a common language and they are smoothly integrable into a visual model. We assume that a user is an experienced (sequential) programmer and therefore writing sequential codes in a textual language is more convenient. We focus on reducing complexity of developing parallel aspects, not developing of sequential parts.

In the current version of Kaira, sequential codes (inserted into visual models) can be written in C++. We are also working on Java and Python versions but it is still in an early state. C++ codes can be inserted into models in two ways. Codes can be inserted into transitions and places, and our inscription language[7] can be enriched by C++ functions and types.

Models created in Kaira can be simulated in the same way as in a modelling tool. A programmer can see what happens inside of an incomplete application without any special debugging tools. Because of our effort to create a practically usable programming environment, Kaira is able to generate standalone parallel applications. The resulting applications use MPI and pthreads as parallel backends, hence these applications can be directly run on HPC hardware.

The other way how to use Kaira is to generate libraries which can be used in other programming environments. Generating libraries allows creating building blocks for other environments. Performance

[5] `http://verif.cs.vsb.cz/kaira`

[6] `http://www.gnu.org/licenses/gpl-3.0.html`

[7] The language used for expressions in Coloured Petri net

demanding parts of an existing application can be gradually replaced by parallel versions designed in Kaira. The use of libraries is not limited to C++ but we can also generate modules for Octave[8], the tool for numerical computations. We are also working on a support for Matlab[9]. Moreover Kaira is able to generate a library with a remote procedure call (RPC) architecture (both client and server are generated). Therefore the user can run the main program on a single computer in a single instance and computationally demanding codes can be transparently executed on a cluster.

Figure 1 shows a basic usage of Kaira on *pingpong* example. We show a comparison of the development process in the case of plain C++ with MPI and in the case of Kaira.

2 Profiling

Here we mention the recently improved feature of Kaira. When the performance of an application matters, then it is very important to see what happens inside it. This kind of performance measurement is called *profiling*. There are matured tools like Scalasca [10][6] or Vampir [11] focused on this area. But we can show that our approach allows simplifying some parts or providing more precise information, because our tool is not a universal tool that can handle any program, but we profile a program with a structure captured by our model. Generally there are three steps during profiling:

1. Instrumentation (i.e. putting measuring codes inside an application)
2. Data collecting
3. Data processing and visualization

In general, it is a non-trivial task to correctly instrument a compiled program. But in our case we generate the whole communication and parallelization codes and therefore we can put measuring codes precisely into interesting parts without any additional effort. Moreover, the whole process is compiler and architecture independent.

In the second step, it is important what is monitored. Collecting too much data deforms a real run of an application. Usually what is measured is adjusted on the level of functions. But there can be a large number of user and internal functions, therefore it is not always easy to setup the right list of monitored ones. In our approach, we use Petri net where the user can specify what is measured in terms of places, transitions and tokens (see Figure 2). Petri nets are also used in the third step, i.e. how to present results to a user. Kaira offers a replay of a program in the form of the "token game" in Petri nets, or statistical summaries like running times of transitions (see Figure 3), number of tokens in a place, etc.

3 Conclusion

As we said, the main aim is to provide a high level tool for the area where we think that such tool is missing. Our approach tries to apply ideas from the world of modelling tools to the area of HPC programming. Kaira is designed to preserve basic attributes of modelling tools like an easy usage, fast prototyping and simulations but provides them in the form of a practically usable environment for creating real-world HPC applications. Therefore important features are generating standalone applications and libraries, ability to run on real HPC hardware and cooperation with commonly used technologies.

[8] `http://www.gnu.org/software/octave/`
[9] `http://www.mathworks.com/products/matlab/`
[10] `http://www.scalasca.org/`
[11] `http://www.vampir.eu/`

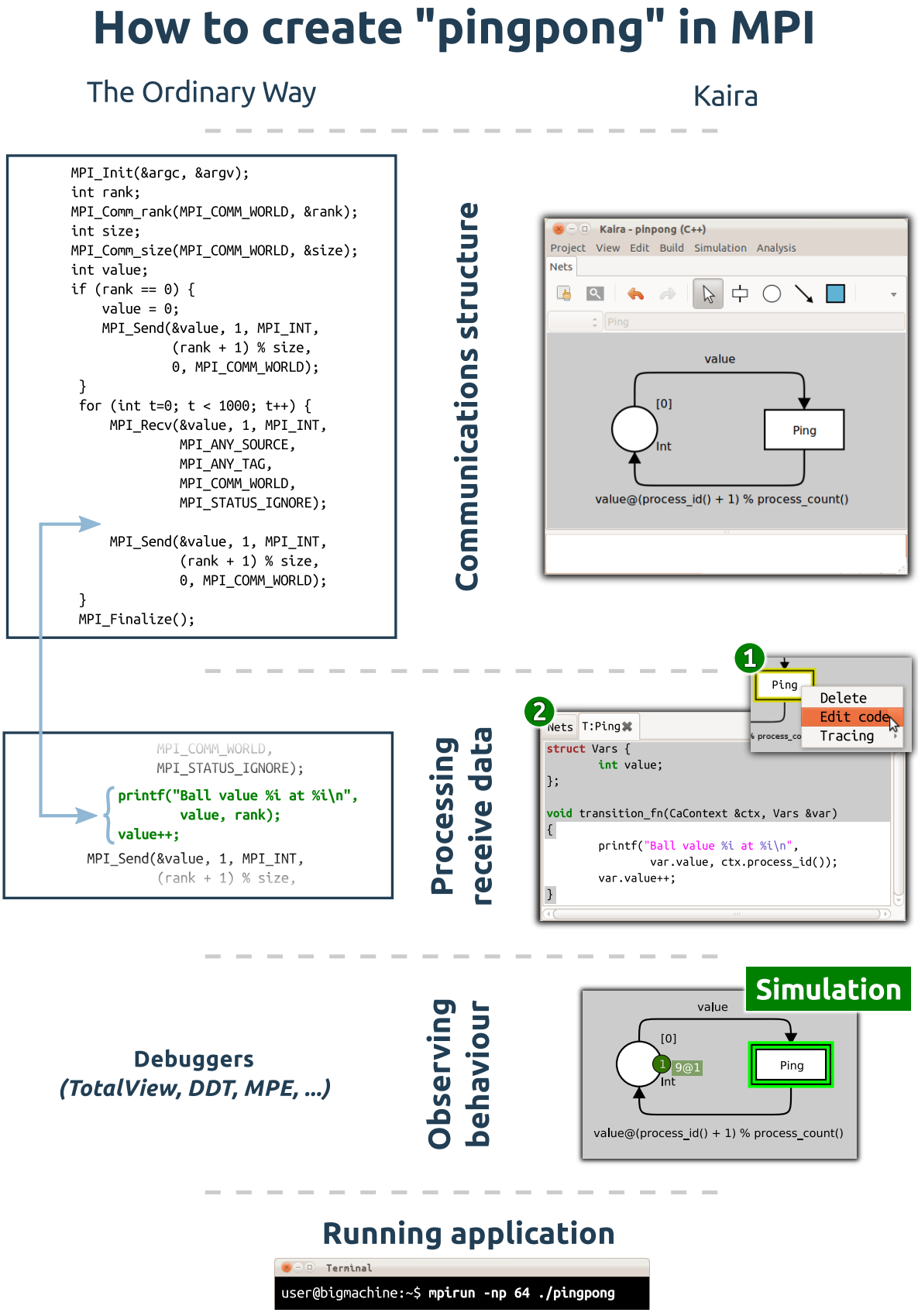


Figure 1: A demonstration of a standard usage of MPI and Kaira

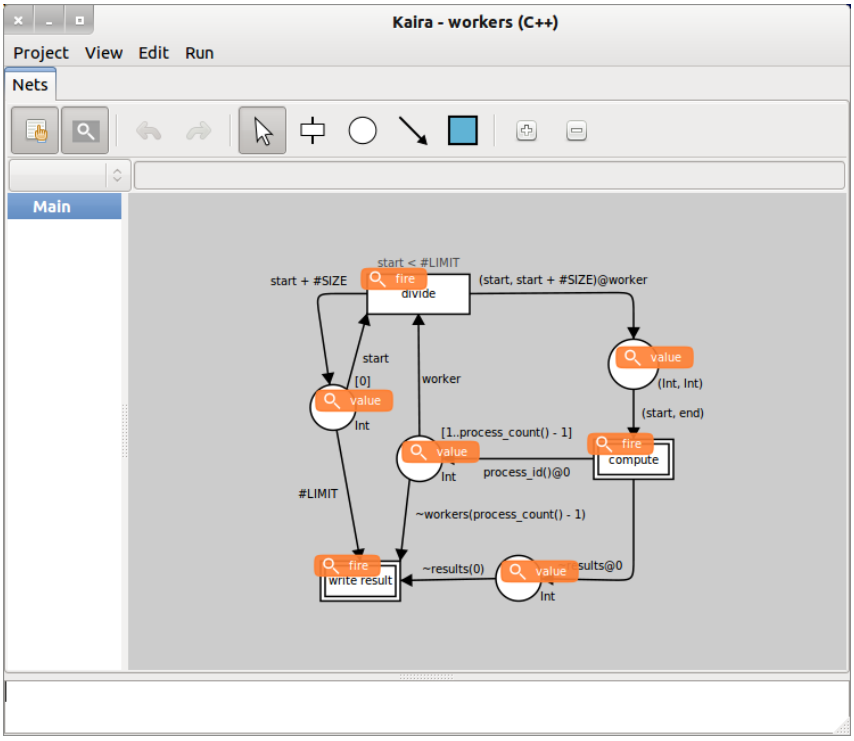


Figure 2: A specification what will be traced (orange marks).

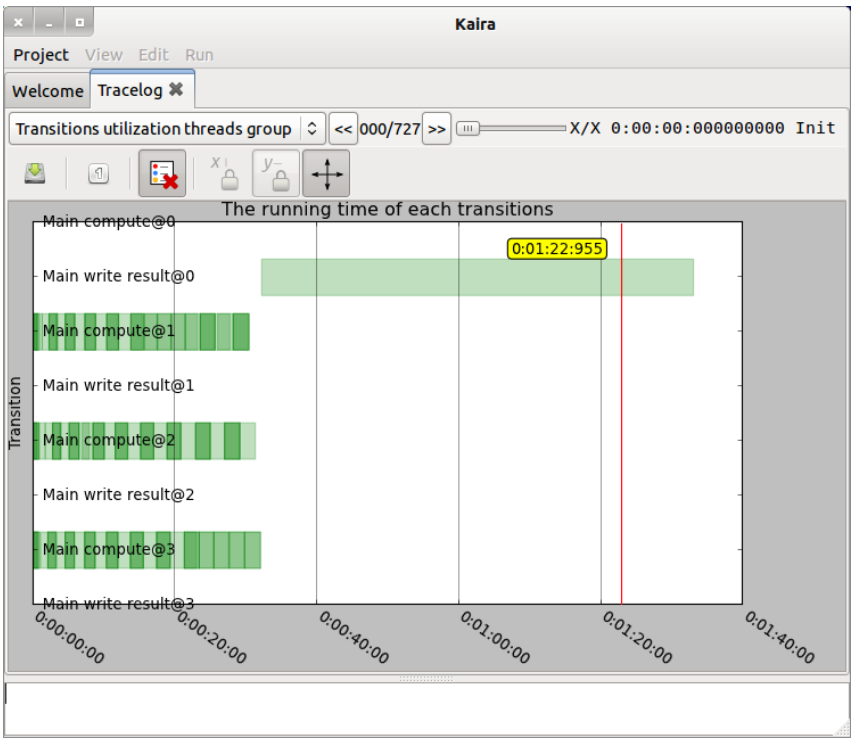


Figure 3: A chart of transitions utilization.

The implementation of Kaira is now in a state where all necessary parts, from visual modelling to debugging produced applications, are functional and the idea of the tool can be practically tested with real programs. On the other hand there are still missing some more advance optimizations. Implementation of optimizations is one of goals in the future. Our research will be focused on problems about data structures, locking mechanisms and a memory management in this topic.

Another part of future works is implementation of some verification techniques. Semantics of our models are too general and most interesting questions are undecidable, but still we can answer some useful questions about of application by a state space analysis, for example checking behaviour on some hardware architectures that is not available for us. With our approach, we do not need to extract a model from source codes because all parallel behaviour is already in form of CPN. Our preliminary results show that this approach can be successfully used to generate the state spaces for small instances of practical applications.

References

[1] Stanislav Böhm and Marek Běhálek. Kaira: Modelling and generation tool based on Petri nets for parallel applications. In *UkSim 13th International Conference on Computer Modelling and Simulation*, pages 403 –408, 30 2011-april 1 2011.

[2] Stanislav Böhm and Marek Běhálek. Generating parallel applications from models based on petri nets. *Advances in Electrical and Electronic Engineering*, 10(1), 2012.

[3] Stanislav Böhm and Marek Běhálek. Usage of Petri nets for high performance computing. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 37–48, New York, NY, USA, 2012. ACM.

[4] Marek Běhálek, Stanislav Böhm, Pavel Krömer, Martin Šurkovský, and Ondřej Meca. Parallelization of ant colony optimization algorithm using Kaira. In *11th International Conference on Intelligent Systems Design and Applications (ISDA 2011)*, Cordoba, Spain, November 2011.

[5] Constantinos T. Delistavrou and Konstantinos G. Margaritis. Survey of software environments for parallel distributed processing: Parallel programming education on real life target systems using production oriented software tools. *Informatics, Panhellenic Conference on*, 0:231–236, 2010.

[6] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[7] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:213–254, 2007.

[8] Lars Michael Kristensen and Michael Westergaard. Automatic structure-based code generation from coloured Petri nets: a proof of concept. In *Proceedings of the 15th international conference on Formal methods for industrial critical systems*, FMICS'10, pages 215–230, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Kristian Bisgaard Lassen and Simon Tjell. Translating colored control flow nets into readable java via annotated java workflow nets.

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

Visibly Pushdown Automata with Multiplicities: Finiteness and K-Boundedness *

Mathieu Caralp Pierre-Alain Reynier
Jean-Marc Talbot
Laboratoire d'Informatique Fondamentale de Marseille, AMU & CNRS, UMR 7279
{mathieu.caralp,pierre-alain.reynier,jean-marc.talbot}@lif.univ-mrs.fr

Abstract

We propose an extension of visibly pushdown automata by means of weights (represented as positive integers) associated with transitions, called visibly pushdown automata with multiplicities.

We prove the problem of deciding whether the multiplicity of an automaton is finite to be in PTIME. We also consider the K-boundedness problem, *i.e.* deciding whether the multiplicity is bounded by K: we prove this problem to be EXPTIME-complete when K is part of the input and in PTIME when K is fixed.

1 Introduction

Visibly pushdown automata (VPA for short) have been proposed in [1] as an interesting subclass of pushdown automata, strictly more expressive that finite state automata, but still enjoying good closure and decidability properties. They are pushdown automata such that the behavior of the stack, *i.e.* whether it pushes or pops, is visible in the input word. Technically, the input alphabet is partitioned into call, return and internal symbols. When reading a call the automaton must push a symbol onto the stack, when reading a return it must pop and when reading an internal it cannot touch the stack. The partitioning of the alphabet induces a nesting structure of the input word. Calls and returns can be viewed as opening/closing brackets, and well-nested words are words where every call symbol (resp. return symbol) has a matching return (resp. call).

The original motivation for their introduction was for verification purposes, the stack being used for the modelization of call/returns of functions. Another application domain is the processing of XML documents. Indeed, unranked trees in their linear form can be viewed as well-nested words. Actually, the model of visibly pushdown automata is expressively equivalent to that of finite tree automata, see [1].

It is quite standard to extend a class of automata with weights, by adding a labeling function assigning a weight to each transition. In this work, we consider VPA with multiplicities ($\mathbb{N}$-VPA for short) where weights are positive integers (multiplicities). The multiplicity of a run is the product of the multiplicities of the transitions used along it. The multiplicity of a word is the sum of the ones of all its accepting runs. Finally, the multiplicity of the automaton is the supremum of the multiplicities of the words it accepts. This model extends the model of finite state automata with multiplicities [9].

The first problem we consider is the finiteness of the multiplicity of an automaton, *i.e.* does there exist $K \in \mathbb{N}$ such that the multiplicity is bounded by K. To solve this problem, we extend a characterization of finite state automata based on patterns to visibly pushdown automata. We also provide an algorithm to

*Partially supported by the ANR Project ECSPER (ANR-09-JCJC-0069).

decide the presence of these patterns in polynomial time. The second class of problems asks whether the multiplicity of an automaton is bounded by K, where K is given. This problem can be considered under the hypothesis that K is part of the input, or is fixed. We show that the problem is EXPTIME-complete in the first case, and can be solved in polynomial time in the second one.

Definitions are given in Section 2. Comparisons with existing results for tree automata with multiplicities are drawn in Section 3. In Section 4, we give the characterization of $\mathbb{N}$-VPA with infinite multiplicity based on original patterns and the decision procedure associated. We study K-boundedness problems in Section 5. This paper is a short version of [5], you can find the details of proofs and results about tree automata within.

2 Visibly pushdown automaton with multiplicities

All over this paper, Σ denotes a finite alphabet partitioned into three disjoint sets Σ_c, Σ_r and Σ_ι, denoting respectively the *call*, *return* and *internal* alphabets. We denote by Σ^* the set of (finite) words over Σ and by ε the empty word. The length of a word u is denoted by $|u|$. The set of *well-nested* words Σ^*_{wn} is the smallest subset of Σ^* such that $\Sigma^*_\iota \subseteq \Sigma^*_{\mathsf{wn}}$ and for all $c \in \Sigma_c$, all $r \in \Sigma_r$, all $u, v \in \Sigma^*_{\mathsf{wn}}$, $cur \in \Sigma^*_{\mathsf{wn}}$ and $uv \in \Sigma^*_{\mathsf{wn}}$. Let $u = \alpha_0 \cdots \alpha_{k-1} \in \Sigma^*$ with $k = |u|$, then $u_{i,j}$ denotes the word $\alpha_i \cdots \alpha_{j-1}$ for $0 \leq i \leq j \leq k$.

Visibly pushdown automata (VPA) [1] are a restriction of pushdown automata in which the stack behavior is imposed by the input word. On a call symbol, the VPA pushes a symbol onto the stack, on a return symbol, it must pop the top symbol of the stack and on an internal symbol, the stack remains unchanged. We introduce the model of VPA with multiplicities in $\mathbb{N}$ ($\mathbb{N}$-VPA for short), by labeling transitions of VPA by positive integers:

Definition 1 ($\mathbb{N}$-VPA)**.** A *visibly pushdown automaton with multiplicities* ($\mathbb{N}$-VPA) over Σ is a tuple $A = (Q, \Gamma, \delta, Q_{in}, Q_f, \lambda)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is the set of initial states, $Q_f \subseteq Q$ is the set of final states, Γ is a finite stack alphabet, $\delta = \delta_c \uplus \delta_r \uplus \delta_\iota$ is the set of transitions, with $\delta_c \subseteq Q \times \Sigma_c \times \Gamma \times Q$, $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\delta_\iota \subseteq Q \times \Sigma_\iota \times Q$ and λ is a labeling function $\lambda : \delta \to \mathbb{N}_{>0}$.

Configuration - Run - Degree of ambiguity. A *configuration* of a $\mathbb{N}$-VPA is a pair $(q, \sigma) \in Q \times \Gamma^*$ (where Γ^* denotes the set of finite words over Γ). We denote by $\perp$ the empty word on Γ. Initial (resp. final) configurations are configurations of the form $(q, \perp)$, with $q \in Q_{in}$ (resp. $q \in Q_f$).

A *run* of A on a sequence of transitions $\{t_i\}_{1 \leq i \leq k}$ from a configuration (q, σ) to a configuration (q', σ') over a word $u = \alpha_0 \dots \alpha_{k-1} \in \Sigma^*$ is a finite non-empty sequence $\rho = \{(q_i, \sigma_i)\}_{0 \leq i \leq k}$ such that $q_0 = q$, $\sigma_0 = \sigma$, $q_k = q'$, $\sigma_k = \sigma'$ and for each $1 \leq i \leq k$, $t_i = (q_{i-1}, \alpha_{i-1}, \gamma_i, q_i) \in \delta_c$ and $\sigma_i = \sigma_{i-1}\gamma_i$ or $t_i = (q_{i-1}, \alpha_i, \gamma_i, q_i) \in \delta_r$ and $\sigma_{i-1} = \sigma_i\gamma_i$, or $t_i = (q_{i-1}, \alpha_i, q_i) \in \delta_\iota$ and $\sigma_i = \sigma_{i-1}$. A run is *accepting* if it starts in an initial configuration and ends in a final configuration. The *degree of ambiguity* of A, denoted by $\mathsf{da}(A)$, is the maximal number of accepting runs for any possible input word.

Language. A word u is accepted by A if there exists an accepting run of A on u. The *language* of A, denoted by $\mathscr{L}(A)$, is the set of words accepted by A. Note that we consider acceptance on empty stack, which implies that all accepted words are well-nested. Unlike [1], we do not consider returns on empty stack and unmatched calls. This assumption is done to avoid technical details but the general case could be handled [1].

[1] More precisely, given a general $\mathbb{N}$-VPA A, one can build a $\mathbb{N}$-VPA A' according to Definition 1 such that accepting runs of A' are in bijection with those of A. This can be achieved by adding self-loops on initial states that allow to push a special symbol (for the returns on empty stack) and self-loops on final states that allow to pop any symbols.

Multiplicity. For each transition $t \in \delta$, $\lambda(t)$ is called the *multiplicity* of t. Let ρ be a run on the sequence of transitions $\{t_i\}_{1\leq i\leq k}$ over the word u and let $m_i = \lambda(t_i)$ for $1 \leq i \leq k$. The multiplicity of ρ, denoted by $\langle\rho\rangle$ is equal to $\prod_{1\leq i\leq k} m_i$. Let a word $u \neq \varepsilon$, we write $(q,\sigma) \xrightarrow{u|m} (q',\sigma')$ when there exists a run over u from (q,σ) to (q',σ') with multiplicity m.

The *multiplicity* of u, denoted by $\langle u\rangle$ is the sum of the multiplicities of the accepting runs over u. The *multiplicity* of an $\mathbb{N}$-VPA A, denoted by $\langle A\rangle$, is defined as $\langle A\rangle = \sup\{\langle u\rangle \mid u \in \mathscr{L}(A)\}$. Let $K \in \mathbb{N}$. We say that A is bounded by K if $\langle A\rangle \leq K$. We say that A is *finite* if we have $\langle A\rangle < +\infty$, and *infinite* otherwise. Note that the degree of ambiguity of a VPA is equal to the multiplicity of the corresponding $\mathbb{N}$-VPA where all the multiplicities of transitions are set to 1.

Trimmed. A configuration (q,σ) is *reachable* (resp. *co-reachable*) if there exists $u \in \Sigma^*$ and $q_0 \in Q_{in}$ (resp. $q_f \in Q_f$) such that $(q_0,\bot) \xrightarrow{u|m} (q,\sigma)$ (resp. such that $(q,\sigma) \xrightarrow{u|m'} (q_f,\bot)$). A VPA A is *trimmed* if every reachable configuration is co-reachable, every co-reachable configuration is reachable and if every state of A belongs to a reachable configuration. In [4], we present a procedure which allows to trim a VPA and which preserves the set of accepting runs.

3 Relating Tree Automata and VPA

There is a strong relationship between words written over a partitioned alphabet and (un)ranked trees. This relationship extends to recognizers with VPA on one side and tree automata on the other side. A polynomial time construction from VPA to tree automata is presented in [1], which can be slightly modified to guarantee the isomorphism of accepting computations [3]. Conversely, it is easy to encode ranked trees as well-nested visible words, and to build from a tree automaton a VPA accepting the encodings and preserving the accepting computations as well.

Note that preserving (accepting) computations implies that the degree of ambiguity of the encoded VPA and of the target tree automaton are the same.

Hence, one may now wonder whether this relationship extends to models with weights and what are the results known for weighted tree automata on the semiring $(\mathbb{N},+,\cdot)$ that carry over $\mathbb{N}$-VPA: this question is crucial as in one direction, it may be the case that problems we want to address could be solved thanks to this relationship and on the other direction, new results for $\mathbb{N}$-VPA may carry over weighted tree automata almost for free.

Let us briefly recap some known results for tree automata with weights/costs. In [13], (ranked) tree automata with polynomial costs are considered over several semirings. However, the result of the computation is the set of costs computed for each accepted run (no combination is made with the accepting computations over the same input tree). These results are extended in [2] by considering more general semirings but without addressing complexity issues.

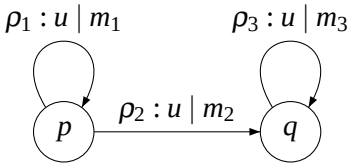
However, the algorithms for finiteness of the degree of ambiguity [11] (deciding $\mathsf{DA} = \mathsf{da}(A) < +\infty$) in PTIME and of the cost of some tree automaton with costs [13] (deciding $\mathsf{MM} = \sup\{\langle\rho\rangle \mid \rho \text{ an accepting computation}\} < +\infty$) in PTIME can be combined to get a PTIME algorithm for finiteness of weighted tree automata, thanks to the following statement : $\max(\mathsf{DA},\mathsf{MM}) \leq \langle A\rangle \leq \mathsf{DA} * \mathsf{MM}$. Thanks to the PTIME encoding of $\mathbb{N}$-VPA into weighted tree automata, we obtain a PTIME algorithm for finiteness of $\mathbb{N}$-VPA. However, our approach provides a direct method based on VPA and a rather intuitive algorithm compared to [11, 13].

4 Characterization and decision of infinite $\mathbb{N}$-VPA

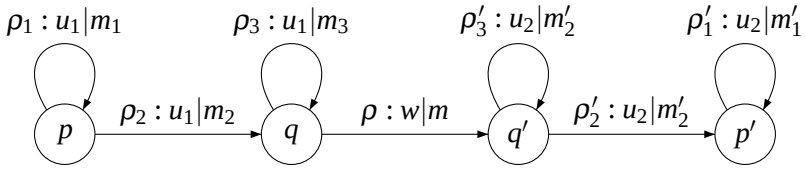
In this section, we give a characterization on $\mathbb{N}$-VPA ensuring their infiniteness by means of patterns. Then, based on this characterization, we devise a PTIME algorithm to solve the finiteness problem.

4.1 Characterization

We introduce the criteria depicted on Figures 1(a) and 1(b) which characterize infinite $\mathbb{N}$-VPA. Pattern of Figure 1(a) coincides with patterns for finite-state automata with multiplicities (see [14, 7]). Pattern of Figure 1(b) is specific to the model of VPA. Intuitively, the loop over a well-nested word is splitted into two loops on words u_1 and u_2, such that the concatenation u_1u_2 is a well-nested word but u_1 is not well-nested. We say that A contains a pattern whenever there exist words in Σ^*, states of A and runs in A that fulfill all the conditions of the pattern.



(a) **(S1)** Well-nested case: $u \in \Sigma^*_{\mathsf{wn}}$. $\rho_1\rho_2\rho_3$ is a run and $m_1 > 1$ or $\rho_1 \neq \rho_2$.



(b) **(S2)** Matched loops case: $w \in \Sigma^*_{\mathsf{wn}}$, $u_1u_2 \in \Sigma^*_{\mathsf{wn}}$, and $u_1 \notin \Sigma^*_{\mathsf{wn}}$. $\rho_1\rho_2\rho_3\rho\rho'_3\rho'_2\rho'_1$ is a run, and either ($m_1 > 1$ or $m'_1 > 1$), or ($\rho_1 \neq \rho_2$ or $\rho'_1 \neq \rho'_2$).

Figure 1: Patterns characterizing infinite multiplicity.

Theorem 2. *Let A be a trimmed $\mathbb{N}$-VPA. A is infinite iff A complies with one of the criteria **(S1)** and **(S2)**.*

The proof of the fact that the presence of one of the criteria implies an infinite multiplicity is easy. The other part of the proof relies on the two technical Lemmas 3 and 4 which we present intuitively. To state these lemmas, we define the constant $N = (n^2L)^2|\Gamma|$ with $n = |Q|$, $L = \max\{\lambda(t) \mid t \in \delta\}$ and the function $\psi : \mathbb{N} \to \mathbb{N}$ as $\psi(z) = n(Nz)^{2^n}$. Pattern **(S1)** allows to increase the multiplicity along a well-nested word. Lemma 3 states that if A does not comply with **(S1)**, then a well-nested word u whose multiplicity is greater than $\psi(l)$ has a well-nested subword v whose multiplicity is greater than l, and such that the height of u is greater than the height of v. Then, Lemma 4 applies iteratively Lemma 3 to prove that a word with large multiplicity has a large height, and hence allows to find pattern **(S2)**, using a vertical pumping.

$h_u(x)$ is the height of u at position x and $s_u(x,y)$ is the sum of the multiplicities of the runs ρ' over $u_{x,y}$ such that there exists an accepting run $\rho\rho'\rho''$ over u.

Lemma 3. *We suppose that A is infinite but A does not comply with **(S1)**. Let $u \in \mathscr{L}(A)$, $l \in \mathbb{N}_{>0}$ and x,y be two positions such that $0 \leq x \leq y \leq |u|$, $u_{x,y} \in \Sigma^*_{\mathsf{wn}}$ and $s^u_{x,y} \geq \psi(l)$. Then there exist two positions $x < x' \leq y' < y$ such that $u_{x',y'} \in \Sigma^*_{\mathsf{wn}}$, $h_u(x') = h_u(x) + 1$ and $s^u_{x',y'} \geq l$.*

Lemma 4. *We suppose that A is infinite but A does not comply with **(S1)**. Then A complies with **(S2)**.*

4.2 Decidability of finiteness

We show in this part how to decide in PTIME the presence of one of the patterns.

The algorithm uses bunches of inference rules applied as a saturation procedure: the first bunch builds a set $\mathscr{S}_1$ of tuples composed of 6 states and a Boolean, which allows to decide the presence of the pattern **(S1)**. The 6 states represent the source and the target of 3 paths over the same well-nested word and the Boolean retains an information about a multiplicity greater than 1 or the fact that different paths are considered. The second bunch builds a set $\mathscr{S}_2$ of tuples composed of 12 states and a Boolean which allows to decide the presence of pattern **(S2)**.
Let $A = (Q, \Gamma, \delta, Q_{in}, Q_f, \lambda)$ be a trimmed $\mathbb{N}$-VPA. We now define the sets $\mathscr{S}_1$ and $\mathscr{S}_2$.

Let $\mathscr{S}_1 \subseteq Q^6 \times \mathbb{B}$ be a set such that $(p_1, q_1, p_2, q_2, p_2, q_3, B) \in \mathscr{S}_1$ if and only if the following property holds:

$$\begin{aligned} \exists u \in \Sigma_{\mathsf{wn}}^* \text{ and three runs } \eta_i : (p_i, \bot) \xrightarrow{u} (q_i, \bot) \text{ of } A \text{ for } i \in \{1,2,3\} \\ \text{such that } B = (\rho_1 \neq \rho_2 \vee \langle \rho_1 \rangle > 1) \end{aligned} \tag{1}$$

Let $\mathscr{S}_2 \subseteq Q^{12} \times \mathbb{B}$ be a set such that $(p_1, q_1, p_2, q_2, p_3, q_3, q_3', p_3', q_2', p_2', q_1', p_1', B) \in \mathscr{S}_2$ if and only if the following property holds:

$$\begin{aligned} \exists w, uu' \in \Sigma_{\mathsf{wn}}^* \text{ and runs } \rho_i : (p_i, \bot) \xrightarrow{u} (q_i, \sigma_i),\ \rho_i' : (q_i', \sigma_i) \xrightarrow{u'} (p_i', \bot) \text{ and} \\ \rho : (q_3, \bot) \xrightarrow{w} (q_3', \bot) \text{ of } A \text{ for } i \in \{1,2,3\} \\ \text{such that } B = (\rho_1 \neq \rho_2 \vee \rho_1' \neq \rho_2' \vee \langle \rho_1 \rangle > 1 \vee \langle \rho_1' \rangle > 1) \end{aligned} \tag{2}$$

We can build these sets in polynomial time using inference rules. Then:

Proposition 5. *For any trimmed $\mathbb{N}$-VPA $A = (Q, \Gamma, \delta, Q_{in}, Q_f, \lambda)$, $(p, q, q, q, p, p, \top) \in \mathscr{S}_1$ with $p, q \in Q$ if and only if A complies with **(S1)**.*

*For any trimmed $\mathbb{N}$-VPA $A = (Q, \Gamma, \delta, Q_{in}, Q_f, \lambda)$, $(p, p, p, q, q, q, p', p', p', q', q', q', \top) \in \mathscr{S}_2$ with $p, p', q, q' \in Q$, if and only if A complies with **(S2)**.*

Theorem 6. *Finiteness for $\mathbb{N}$-VPA is in* PTIME.

5 Deciding K-bounded multiplicity

We consider here the K-bounded multiplicity problem: For a given trimmed $\mathbb{N}$-VPA A and an integer K, we ask whether $\langle A \rangle < K$. K can be fixed or be a part of the input.

Theorem 7. *Given a trimmed $\mathbb{N}$-VPA A and $K \in \mathbb{N}_{>0}$, the problem of determining whether $\langle A \rangle < K$ is* EXPTIME*-complete.*

Theorem 8. *Fix $K \in \mathbb{N}_{>0}$. For a trimmed $\mathbb{N}$-VPA A, deciding whether $\langle A \rangle < K$ is in* PTIME.

The first result is an extension of an algorithm described in [6] and the second is an extension of an algorithm described in [14]. These algorithms decide if the ambiguity of a finite state automaton is less than a given integer.

6 Conclusion

In this paper we have presented three differents results : a characterisation of infinite multiplicity of $\mathbb{N}$-VPA, a polynomial algorithm for the finiteness problem and two different algorithms for the K-bounded multiplicity problem (depending whether K is a part of the input or not).

There exists a polynomial construction to build a weighted tree automaton from a $\mathbb{N}$-VPA. Since this construction preserves the multiplicity (we have a bijection between the runs of the two automata), we can easily adapt all the results from $\mathbb{N}$-VPA to weighted tree automaton, and thus acquire a new characterisation for the infinite cost of weighted tree automata on the semiring $(\mathbb{N}, +, \cdot)$. See [5] for more details.

We recall that degree of ambiguity of a word is a special case of multiplicity. The class of finitely ambiguous automata has been investigated for both automata on words and on trees [6, 14, 11, 12]. The interest in this class arised from the fact that it allows an efficient (polynomial) equivalence check. An analogy can be drawn with the context of transducers where the equivalence problem is decidable for finite-valued transducers (and undecidable in general). In [10], the characterization of automata whose multiplicity is finite is used to build a characterization of finite-valued word transducers. We hope this present work will be a first step towards the characterization of finite-valued visibly pushdown transducers, which is a relevant issue as this model is incomparable with bottom-up tree transducers (see [8]).

References

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. STOC '04*, pages 202–211, 2004.

[2] B. Borchardt, Z. Fülöp, Z. Gazdag, and A. Maletti. Bounds for tree automata with polynomial costs. *Journal of Automata, Languages and Combinatorics*, 10(2/3):107–157, 2005.

[3] M. Caralp. Automates à pile visible : ambiguité et valuation. Master's thesis, Aix-Marseille Université, 2011.

[4] M. Caralp, P.-A. Reynier, and J.-M. Talbot. A polynomial procedure for trimming visibly pushdown automata. Technical Report hal-00606778, HAL, CNRS, France, 2011.

[5] M. Caralp, P.-A. Reynier, and J.-M. Talbot. Visibly pushdown automata with multiplicities: finiteness and k-boundedness. In *Proceedings of the 16th international conference on Developments in Language Theory*, DLT'12, pages 226–238, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] T.-H. Chan and O. H. Ibarra. On the finite-valuedness problem for sequential machines. *Theoretical Computer Science 23*, pages 95–101, 1983.

[7] R. De Souza. *tude structurelle des transducteurs de norme borne*. PhD thesis, ENST, France, 2008.

[8] E. Filiot, J.-F. Raskin, P.-A. R. Reynier, F. Servais, and J.-M. Talbot. Properties of visibly pushdown transducers. In *Proc. MFCS'10*, volume 6281 of *LNCS*, pages 355–367. Springer, 2010.

[9] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[10] J. Sakarovitch and R. de Souza. On the decidability of bounded valuedness for transducers. In *Proc. MFCS'11*, volume 5162 of *LNCS*, pages 588–600. Springer, 2008.

[11] H. Seidl. On the finite degree of ambiguity of finite tree automata. *Acta Inf.*, 26(6):527–542, 1989.

[12] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437, 1990.

[13] H. Seidl. Finite tree automata with cost functions. *Theor. Comput. Sci.*, 126(1):113–142, 1994.

[14] A. Weber and H. Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991.

Runtime Verification for Real-Time Automotive Embedded Software

Sylvain Cotard
LUNAM Université. Université de Nantes.
IRCCyN UMR CNRS 6597
Nantes, France.
Renault, SAS
sylvain.cotard@irccyn.ec-nantes.fr

Abstract

We present the design of an error detection service for real-time automotive embedded software. The service monitors at runtime the data flows in a graph of communicating real-time tasks. At design-time, monitors are automatically generated from formal models; at compile-time, monitors are embedded in the Real Time Operating System (RTOS) kernel; at runtime, errors are detected and notified with a small and deterministic latency.

1 Introduction

During the past 15 years, the number of services provided in vehicles caused the evolution of Electrics and Electronics (E/E) systems from federated architectures (one function per Electronic Control Unit (ECU)) to integrated architectures (several functions per ECU). In this context, automotive OEMs (Original Equipment Manufacturers) and suppliers are turning toward real-time and multitask-capable operating systems for improved code quality and efficiency. To face new challenges induced by theses changes, automotive industry stakeholders are working on the design of a common architecture supported by standardized software services: AUTOSAR (AUTomotive Open System ARchitecture) [1].

The context of our work is the dependable design of AUTOSAR systems. Among the attributes of dependability, we focus here on software fault tolerance and more specifically on error detection. We are developing an error detection service based on runtime verification in AUTOSAR-like systems [5].

The paper is organized as follows. In section 2 we present the main motivations of our work. In section 3, we recall the object of runtime verification and expose the runtime verification technique of LTL formula from bauer2011. In section 4, we explain how to use runtime verification efficiently in the context of automotive embedded systems. In section 5, we conclude.

2 Motivations

Modern automotive embedded software applications are composed of communicating real-time tasks. Their global behavior depends on many design time and runtime parameters. As an illustration, let us consider the model described in Figure 1. In this example, three concurrent tasks T_0, T_1 and T_2 communicate through two shared buffers b_0 and b_1. T_2 reads data from both b_0 and b_1 to make a coherency check between the input and the output of T_1. A correctness requirement for this application could be: *when T_2 starts reading, the buffers are synchronized and stay synchronized until it has finished*. The buffers are synchronized if the data currently stored in b_1 has been produced with the data currently stored in b_0.

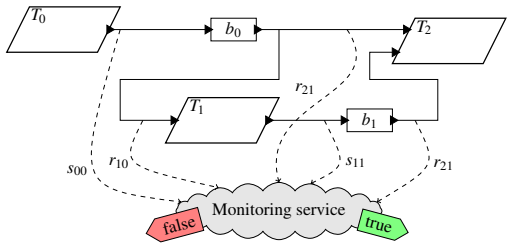


Figure 1: Monitoring architecture. s_{xy} are sending events by T_x to b_y and r_{xy} are receiving events by T_x from b_y.

The satisfaction of such a property depends on parameters such as task scheduling parameters (period, priority, execution times), synchronisation and communication protocols, task assignment to cores (for multicore architecture), etc.

In an integrated architecture, several concurrent real-time applications are hosted on a single execution platform. Today, these platforms are based on monocore microcontrollers. In the near future, multicore microcontrollers (i.e. the combination of two or more calculation units on a same die, that run in parallel) will be used because they offer undeniable advantages in terms of performance and power consumption. Unfortunately, real-time parallel programming is known to be very difficult.

It is thus reasonable to consider that errors will occur at runtime in tomorrow's automotive embedded software. Then, runtime mechanisms to detect and mitigate these errors must be proposed.

A well known solution is to rely on diversification. Our proposal is a specific form of diversification, targeting data flow errors in real-time multitask software systems. Here, we focus on the error detection part. At design time, the expected behaviors of the data flows are specified. They are used with a formal model of the system to generate specialized monitors. The tool *Enforcer*[1] has been built for this purpose. Then, at compile-time, the generated monitors are embedded in the RTOS kernel. Lastly, at runtime, the error detection service uses the monitors to report the behaviors that do not conform to the specification.

3 Runtime Verification

3.1 From Model Checking to Runtime Verification

Runtime Verification (RV) is a lightweight formal method that shares some concepts with *Model Checking* (MC). Both methods ask the designer to specify the properties ϕ that the system should verify. These properties are typically expressed with a temporal logic such as LTL (for RV or MC) or CTL (for MC). In MC, the designer must also provide a model M of the system, typically in the form of a transition system. Then, the model checker solves the problem $M \models \phi$. The answer is either yes, or a counter-example. MC allows to detect design errors.

In RV, the property ϕ is used to generate an event-based monitor that is translated into code to decide at runtime $\sigma \models \phi$, where σ denotes the ongoing execution of the system. The designer must provide some extra information to recognize and preprocess the events of interest. When the monitor receives an event, it outputs a verdict: $true$ (all the possible continuations of the execution will be accepted), $false$ (none of the continuations of the execution will be accepted), or *inconclusive* (some continuations

[1] *Enforcer* is developed by the group *systèmes temps réel* at IRCCyN. It is distributed under GPL licence. It is available here: `http://enforcer.rts-software.org`

will be accepted, some others will not). The monitor must be built such that it outputs its verdict as soon as possible. Runtime verification allows to detect errors that are activated at runtime. That runtime verification can be introduced in industrial real-time embedded systems with a minimal execution time overhead and an acceptable memory footprint as shown in [4].

3.2 Runtime Verification of LTL formulae

3.2.1 Linear Temporal Logic (LTL)

Temporal logics are mathematical tools that deal with the temporal behaviors of discrete event systems. LTL (Linear Temporal Logic) is a temporal logic proposed by Pnueli for the formal specification and verification of reactive systems [8]. LTL formulae express properties about the running of such systems. LTL extends propositional logic with two modalities: X (for neXt) and U (for Until), presented below.
Syntax: Let AP be a set of atomic propositions. The set of LTL formulae over AP is defined inductively as follows: if $p \in AP$, then p is a LTL formula; if ϕ and ψ are LTL formulae then $\neg\phi$, $\phi \wedge \psi$, $\phi U \psi$ and $X\phi$ are also LTL formulae.
Semantics: Let $\Sigma = 2^{AP}$ and $\sigma = s_0 s_1 s_2 \ldots s_i \ldots \in \Sigma^\omega$ an ω-word on Σ. Let $\sigma(i) = s_i$ the i^{th} element of σ, and $\sigma_i = s_i s_{i+1} \ldots$ the suffix of σ starting at the i^{th} element. Let $p \in AP$ and ϕ and ψ two LTL formulae over AP. The satisfaction relation $\sigma \models \phi$ is defined inductively as follows: $\sigma \models true$; $\sigma \models p$ iff $p \in \sigma(0)$; $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$; $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$; $\sigma \models X\phi$ iff $\sigma_1 \models \phi$ (ϕ will be true at the next step); $\sigma \models \phi U \psi$ iff $\exists j \in \mathbb{N}$ s.t $\sigma_j \models \psi$ and $\forall k < j, \sigma_k \models \phi$ (ϕ remains true until ψ becomes true).

From these basic operators, it is possible to define other logical operators ($\vee$, $\Rightarrow$, ...) and modalities such as F (a property will eventually be true) and G (a property is and will always be true).

3.2.2 Automatic Generation of Monitors

We recall here the construction of an RV monitor for the LTL formulae proposed in [2]

The monitor is given in the form of a Moore machine. The input alphabet of the machine is the set 2^{AP} where AP is the set of atomic propositions used to write ϕ. The output alphabet is the set $\mathbb{B}_3 = \{\top, \bot, ?\}$ (resp. true, false and inconclusive). The procedure to build the machine is illustrated by figure 2. It is composed of two similar branches. The top branch builds a monitor that outputs either $\top$ or ? for formula ϕ. The bottom branch is the same for formula $\neg\phi$ so that the outputs of the monitor can be interpreted as $\bot$ and ?.

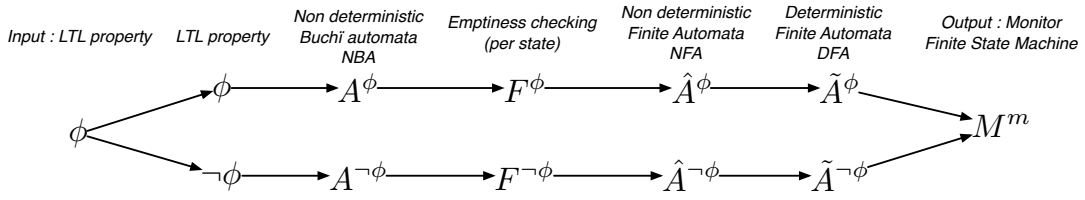


Figure 2: Procedure to build a Moore machine from LTL formulae [2]

Consider the formula $\phi = G\,a$ (a is and will always be true). The first step consists in computing two non-deterministic Büchi automata (NBA) accepting the same $\omega-$languages as ϕ and $\neg\phi$. This can be done by using for instance the technique described by Gastin and Oddoux in [7]. The result is given on Table 1 (left side).

The next steps consist in computing two deterministic finite automata (DFA) that recognize the languages of the prefixes of the ω-word accepted by the two NBA. This requires to perform two classical operations: first, emptiness checking for each state of each NBA; then, determinization of finite automata. The result is given on Table 1 (middle). It is worth noting that a trap state has been added during the determinization step of ϕ so that the resulting automata is complete with regards to 2^{AP}.

The Moore machine is finally built by a synchronized product of the two DFA. The output function is then computed to associate a verdict $\top$, $\bot$ or ? to each state. The result is given on Table 1 (right side).

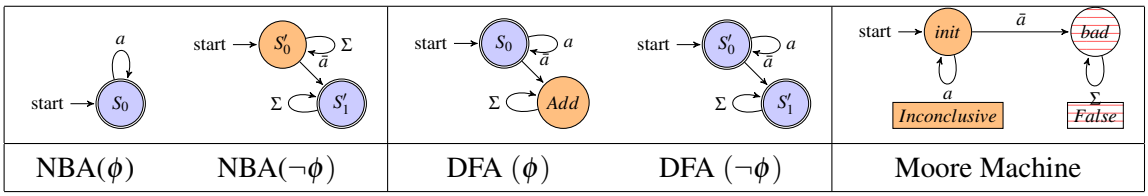


Table 1: Procedure for the Moore Machine synthesis (monitor for a given LTL formula)

RV of LTL formulae has some limits. Indeed, there exist LTL properties for which it is not possible to build a monitor that outputs $\top$ or $\bot$ in finite time. For these formulae, the procedure builds a Moore machine with a single state that outputs ?. Detailed discussion on the class of properties that can be monitored can be found in [6] and [2]. In [2], Bauer *et al.* describe an experiment based on a set of patterns commonly found in the specification of concurrent and reactive systems. On 107 patterns expressed as valid LTL formulae, 53 are found monitorable. This tends to show that, despite its limits, this technique can be used for a broad range of properties.

4 Towards an Error Detection Service based on Runtime Verification

RV appears to be a promising approach to generate monitors that could be used by an error detection service. This service should implement the following functions: identification of the event of interest ; update of the set of atomic properties that are true after the occurrence of an event ; computation of the output of the monitor to this update ; if the output is either $\top$ or $\bot$, notification to the error mitigation component. We have developed such a service (and the companion toolchain) for an AUTOSAR-like platform based on the Trampoline RTOS.

4.1 Implementation Constraints in Automotive Embedded Systems

A typical automotive embedded system is hosted on a microcontroller with limited resources (from 32KB to 1MB of RAM, a few MB of Flash, processor frequency under 100MHz) and must fulfill real-time constraints (required response time for some motor control functions is less than 1 ms). These constraints must be taken into account in the design of the error detection service. In other words, the service must accomplish its functions with a small and predictable overhead (both in time and memory) and it must also offer a predictable response time (ie. the time between the date of the occurrence of the event which allows to deduce that the property is verified/violated and the date of the notification of this deduction to the error mitigation component must be bounded).

As we are targeting the automotive domain, the characteristics of AUTOSAR must also be taken into account. One important characteristic is the static nature of AUTOSAR: all software objects are created at compile-time. A direct consequence is that all objects are known a priori. This allows to use specialization to achieve low and deterministic overhead for system services.

4.2 Efficient Identification and Preprocessing of the Events

To minimize the amount of runtime computation required to identify and preprocess the events, we have taken the following design decisions:

- the events can only be system calls. The set of system calls to intercept are identified offline. The identification consists in a triplet (called function, caller id, parameter values) ;
- the preprocessing of the events is also performed offline.

The first decision allows to automatically inject the event identification code in the source code of the RTOS kernel. The identification consists in a lookup in a table. This is done in $O(1)$.

To realize the second decision, we must bridge the gap between the intercepted events and the atomic proposition used to write the monitored properties. This is done with a deterministic finite automaton (DFA) that models the monitored system, and a labeling function that decorates the state of this DFA with atomic propositions.

Formally, the DFA A^s over alphabet Σ^s is defined as $A^s = (Q^s, i^s, \rightarrow_s)$ where Q^s is the finite set of states, $i^s \in Q^s$ is the initial state and $\rightarrow_s \subset (Q^s \times \Sigma^s) \mapsto Q^s$ is the transiton function.

The type of the labeling function is given by $\lambda^s \subset Q^s \mapsto 2^{AP}$.

The monitor computed by the RV technique is given by $M^m = (Q^m, i^m, \rightarrow_m, \gamma^m)$ where Q^m is the finite set of states, $i^m \in Q^m$ is the initial state, $\rightarrow_m \subset (Q^m \times 2^{AP}) \mapsto Q^m$ is the transition function and $\gamma^m \subset Q^m \mapsto \mathbb{B}_3$ is the output (injective) function.

To preprocess the events, we compute the Moore machine M' offline over Σ^s defined as $M' = (Q', i', \rightarrow, \gamma')$ where $Q' = Q^s \times Q^m$, $i' = (i^s, i^m)$, $\rightarrow \subset (Q' \times \Sigma^s) \mapsto Q'$ where $(q^s, q^m) \xrightarrow{\sigma} (r^s, r^m)$ iff $q^s \xrightarrow{\sigma}_s r^s$ and $q^m \xrightarrow{u}_m r^m$ and $u \subseteq \lambda^s(r^s)$ and $\gamma^m(q^m) = ?$, and $\gamma' \subset Q' \mapsto \mathbb{B}_3$ where $\gamma'(q^s, q^m) = \gamma^m(q^m)$.

Notice that we do not build the transitions outgoing from a state that outputs either $\bot$ or $\top$. When such a state is reached, the work of the monitor is finished. In practice we build only the subset of Q' composed of reachable states with a depth-first exploration starting at (i^s, i^m).

The machine M' reacts directly to the intercepted events. If the machine is encoded with a matrix, this reaction consists in another lookup in a table and can be done in $O(1)$.

The update of machine M' is performed after the identification step. Both steps being in $O(1)$, the time overhead is deterministic. We have performed experiments to confirm that this overhead is small enough (see [4]). The system is static, so the memory overhead can be estimated offline. In our experiments, we have also confirmed that the memory overhead is compatible with the constraints of automotive embedded systems.

Lastly, to ensure a small and predictable response time, all the steps are executed in kernel mode, ensuring freedom of interference from application tasks or interrupts.

4.3 Example

Let us consider a system composed of two tasks communicating through a blackboard. For this system, we want to monitor the property *a message written in the buffer is always read before being overwritten*. Let the atomic proposition a denote "the buffer does not contain a message that has not been read and that has been overwritten". Then we have $\phi = G\, a$.

To preprocess the event, we can use an abstract model of the system A^s over the alphabet $\Sigma^s = \{SendMessage, ReceiveMessage\}$ that counts the number of successive occurrences of *SendMessage* in the set $\{0, 1, +\}$. The labelling function is $\{(0 \mapsto a), (1 \mapsto a), (+ \mapsto \bar{a})\}$.

The monitor generated for $G\, a$ has already been given in table 1. The machine M', resulting from the construction explained above, is given in figure 3.

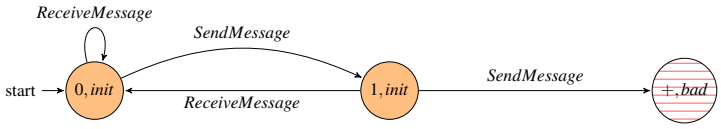


Figure 3: Final monitor

4.4 Tool Support and Integration in Trampoline

We have developed a tool named *Enforcer*, that implements the generation of the machine M' [5]. *Enforcer* processes a model A^s and a property ϕ. It outputs the source code (in C) of the machine M' that is statically injected in the kernel of the Trampoline RTOS [3]. The input language of *Enforcer* allows us to describe rules. Each rule contains a model of a part of the system and a property. The model is defined as a set of Deterministic Finite Automata (DFA) that are then composed with a synchronized product (A^S). The property is expressed in LTL over the set of state of the model.

The tool calls LTL2BA [2] to compute $NBA(\phi)$ and $NBA(\neg\phi)$ and implements all the other steps.

5 Conclusion

Our work aims at providing error detection and mitigation components for future real-time automotive embedded systems. We have designed a service for error detection. This service uses monitors that are automatically generated from formal models thanks to runtime verification techniques.

References

[1] AUTOSAR. http://www.autosar.org. Technical report, AUTOSAR GbR, February 2012.

[2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), 2010.

[3] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline - an open source implementation of the OSEK/VDX RTOS specification. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 62–69, Prague, Tchèque, République, September 2006. IEEE.

[4] Sylvain Cotard, Sébastien Faucou, and Jean-Luc Béchennec. A dataflow monitoring service based on runtime verification for. autosar os: Implementation and performances. In *Proceedings of the 8th annual workshop on Operating Systems Platforms for Proceedings of the 8th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 46–55, 2012.

[5] Sylvain Cotard, Sébastien Faucou, Jean-Luc Béchennec, Audrey Queudet, and Yvon Trinquet. A dataflow monitoring service based on runtime verification for. autosar. In *International Conference on Embedded Software and Systems (ICESS)*, 2012.

[6] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *International Workshop on Runtime Verification (RV)*, 2009.

[7] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *International Conference on Computer Aided Verification (CAV)*, pages 53–65, 2001.

[8] Amir Pnueli. The temporal logic of programs. *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[2] *LTL2BA* is developped by D. Oddoux and P. Gastin. It is available here: `http://www.lsv.ens-cachan.fr/gastin/ltl2ba` The output format of has been modified for our needs. The modified version is included in the *Enforcer* package.

Model Checking Dynamic Distributed Systems

Aiswarya Cyriac
LSV, ENS Cachan, France
cyriac@lsv.ens-cachan.fr

Abstract

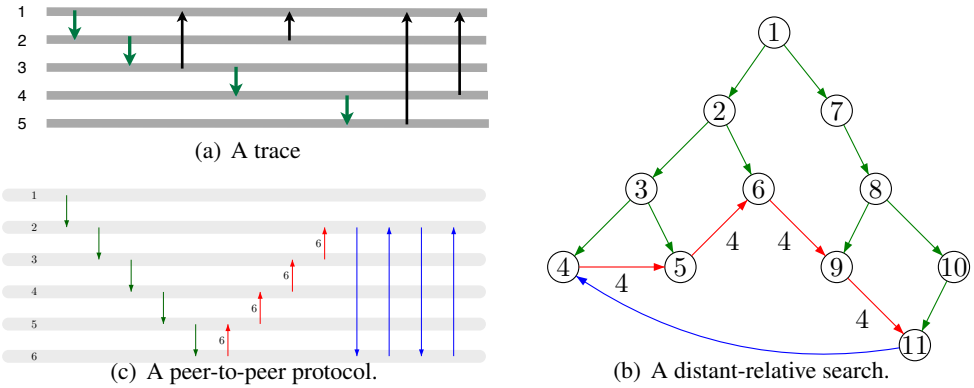
We consider distributed systems with dynamic process creation. We use data words to model behaviors of such systems. Data words are words where positions also contain some data values from an infinite domain. The data values are seen as the process identities. We use automata with a stack and registers to model a distributed system with dynamic process creation. The non emptiness checking of these automata is NP-Complete. While satisfiability of first order logic over data words is undecidable, we show that model checking such automata against full MSO logic (with data equality and comparison predicates) is decidable.

1 Introduction

Distributed systems with a pre-defined finite set of processes have been studied extensively. However, verification of distributed systems with unbounded set of processes or those with dynamic process creation has received relatively little attention. One reason might be the additional difficulty in modeling and model checking caused by the unbounded set of processes. Unfortunately most of the distributed systems we encounter in our everyday life, like internet, creates processes dynamically. Hence verification of distributed systems with dynamic process creation has become a necessity, needless to say it is interesting in its own with the scope of extending the frontiers from bounded number of processes to a dynamic setting.

Verification of systems with dynamic process creation was considered in [3]. Grammars were used to model such systems, and showed that model checking these grammars against MSO is decidable. In [1], a powerful automaton model with stacks and registers are used to model dynamic distributed systems. This extended abstract is an extract from [1], restricting the automaton to use only one stack. We study the non-emptiness problem of these automata. We argue that model checking these automata against MSO with data comparison test (as opposed to equality test which is shown decidable in [1]) is decidable. Our automaton must be seen as a low-level specification formalism rather than an implementation model. This is a first step towards synthesizing local implementations from a global specification, in the spirit of [2] where the authors show how to synthesize the local implementations from a global grammar specification.

This extended abstract is organized as follows. Section 2 defines data words formally and shows how to model distributed protocols as data words. It also introduces the specification language – MSO logic with data comparison. Section 3 introduces the automata formalism and states the results. We conclude in next section with a brief discussion.



(a) A trace

(c) A peer-to-peer protocol.

(b) A distant-relative search.

Figure 1: Data-words for DDS behaviors

2 Data words to model protocols

Notation Given a ranked alphabet A with a mapping $arity : A \mapsto \mathbb{N}$ and a (potentially infinite) set B, we denote by A_B the set $\{a(p_1,\ldots,p_n) \mid a \in A,\ n = arity(a) \text{ and } p_i \in B\}$.

We fix the set of process identifiers of any Dynamic Distributed Systems (abbreviated as DDS henceforth) to be the set of positive integers $\mathbb{N}$ for the scope of this extended abstract.

We consider DDS capable of performing two types of atomic events: create, and message. Each of these events have two participating processes – for create, we have the creating process and the created process; for message, we have the sender and the receiver. The created process will be a fresh (non existing) process, while the creating process as well as the sender and the receiver are assumed to be existing. We also assume the pid of the created process to be bigger (in the natural order) than that of any existing process. This is in accordance with the pid assigning conventions in Unix, thereby facilitating to determine which process is more recent by looking at its process ids. In addition, a message can have the message contents, which we denote by a predefined message type and the set of (existing) process ids appearing in the message. The number of process ids that appear in a message is determined by the message type.

Let $\mathsf{Messages} = \{a,b,\ldots\}$ be the finite set of predefined message types. Then we let $\mathtt{Events} = \{\mathtt{create}, a, b, \ldots\}$ to be the finite ranked alphabet of the DDS with $arity : \mathtt{Events} \to \mathbb{N}$, the arity function. We have $arity(\mathtt{create}) = 2$ and $arity(a) \geq 2$ for $a \in \mathsf{Messages}$. In fact $arity(a) - 2$ gives the number of process ids that appear in the message of type a. An event is an element of $\mathtt{Events}_{\mathbb{N}}$.

A sequence of events gives a behavior of a DDS. For example,

Example 1. The behavior of a DDS depicted in Figure 1(a) is
$trace = \mathtt{create}(1,2)\mathtt{create}(2,3)\mathtt{msg}(3,1)\mathtt{create}(3,4)\mathtt{msg}(2,1)\mathtt{create}(4,5)\mathtt{msg}(5,1)\mathtt{msg}(4,1)$. □

A data word, over a finite ranked alphabet A and an infinite data domain D, is a sequence of elements from A_D. If a position i of a data word is labeled by a letter of *arity* m, we denote the kth data value at i by $data_k(i)$ for all $k \leq m$. For example, if i is labeled by $\sigma(d1,\ldots dn)$, then $data_k(i) = d_k$. Notice that behaviors of DDS are data words over $\mathtt{Events}$ and $\mathbb{N}$. But any data word over $\mathtt{Events}$ and $\mathbb{N}$ need not have an interpretation as the behavior of a DDS. For e.g. $\mathtt{create}(1,2)\mathtt{create}(2,1)$ is a valid data word, but it cannot be seen as the behavior of a DDS since an existing process cannot be created. $\mathtt{create}(2,1)$ is also not a valid behavior, since the pid of the newly created process needs to be bigger than the existing processes.

It might be a bit confusing to see that we have used sequences to represent the behaviors of a DDS. This looks like it captures only *linearzations* of the distributed behavior. However, as we will shortly see, the specification language we use is powerful enough to recover the concurrency information from a linearization. Thus, language of data words $L \subseteq \texttt{Events}^*_{\mathbb{N}}$ can represent a set of behaviors of a DDS, or a protocol. The following example describes peer to peer communication in a DDS.

Example 2. The set of data words of the form $\texttt{create}(1,2)\texttt{create}(2,3)\ldots\texttt{create}(n-1,n)\texttt{req}(n,n-1,n)\texttt{req}(n-1,n-2,n)\ldots\texttt{req}(m-1,m,n)\texttt{msg}(m,n)\texttt{msg}(n,m)\texttt{msg}(m,n)\texttt{msg}(n,m)\ldots\texttt{msg}(m,n)\texttt{msg}(n,m)$ with $m < n$ is a dynamic peer to peer protocol. The informal description of the protocol is as follows. There is a creation phase in which the processes are created in a cascade fashion. After that the last created process requests its parent to be a peer. It can either accept, or refuse by passing the request on to its parent. The messages in the request phase needs to carry the identity of the requesting process in its contents. Figure 1(c) shows this behavior as a distributed system with $n = 6$ and $m = 2$. The data word is obtained by writing down the events in the left to right order. □

Example 3. Consider a DDS which creates processes to form a tree architecture like in Figure 1(b). This word can be represented by a *depth-first-search* listing of the create events. For e.g. in Figure 1(b) it is $\texttt{create}(1,2)\texttt{create}(2,3)\texttt{create}(3,4)\texttt{create}(3,5)\texttt{create}(2,6)\texttt{create}(1,7)\texttt{create}(7,8)\texttt{create}(8,9)\texttt{create}(8,10)\texttt{create}(10,11)$ which we denote by *tree*. This can be followed by a request propagating from the leftmost leaf to the right most leaf only through the leafs (the request message scans the yield of the tree from left to right). This is similar to the seeking phase in the peer-to-peer protocol. A data word for this phase in the example is *seek* $= \texttt{req}(4,5,4)\texttt{req}(5,6,4)\texttt{req}(6,9,4)\texttt{req}(9,11,4)$ Finally, the rightmost leaf (peer) sends a message directly to the leftmost leaf ($\texttt{msg}(11,4)$). Thus a data word representation of Figure 1(b) is *tree*$\,$*seek*$\texttt{msg}(11,4)$.

This example can be seen as modeling the search for a distant relative in a social network. The green part of the tree shows the family tree. The leaves are the current generation. The leaves only know their closest relatives in the current generation (their left and right neighbors in the left-to-right ordering of the leaves). A person in the present generation (process 4) wants to find a kin peer. The request for such a peer must be propagated along the current generation (older generations are perhaps dead). □

Now we describe a powerful language to reason about the properties of data words. We use an extension of MSO over words to data words which allows comparison of data values.

Monadic second order logic over data words We assume countably infinite supplies of first-order and second-order variables. We let $x, y, \ldots$ denote first-order variables, which vary over word positions, and we use $X, Y, \ldots$ to denote second-order variables, which vary over sets of positions.

Definition 4 (MSO logic over data words)**.** The class $\text{MSO}_d(\texttt{Events})$ of *monadic second-order (MSO) formulas* over data words is given by the following grammar, where a ranges over $\texttt{Events}$, and k, ℓ are at most the maximum rank of any letter in $\texttt{Events}$:

$$\varphi ::= a(x) \mid d^{<}_{k,\ell}(x,y) \mid d^{=}_{k,\ell}(x,y) \mid x \leq y \mid x \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists x\phi \mid \exists X\phi$$

If the free variable x is interpreted as position i of a data word, then Formula $a(x)$ holds if the label of i is a. If the free variable x and y are interpreted as positions i and j respectively, Formula $d^{=}_{k,\ell}(x,y)$ holds if k and ℓ are at most the rank of the letters labeling i and j respectively, and $data_k(i) = data_\ell(j)$. Semantics of Formula $d^{<}_{k,\ell}(x,y)$ is similar but requires $data_k(i) < data_\ell(j)$ instead of $data_k(i) = data_\ell(j)$. Formula $x \leq y$, the boolean connectives, and quantifiers are self-explanatory. We may use the usual abbreviations $x < y$, $\forall x\phi$, $\phi \rightarrow \psi$ and so on. If ϕ is a sentence, i.e., it does not have any free variable, then we set $L(\phi)$ to be the set of data words w such that $w \models \phi$.

Example 5. Consider the property that any process which requests for a peer eventually gets a peer. This can be said by the following formula: $\forall x\, req(x) \rightarrow \exists y(y > x \wedge msg(y) \wedge d^{=}_{3,2}(x,y))$. That is, if there is a "*req*" event, then there is a "*msg*" event in the future such that the parameter of "req" event and the receiver of the "msg" event are the same.

Example 6. Consider a property that the participants of any message are always leaves, i.e., they do not create other processes. This can be said by the formula
$\forall x \neg \texttt{create}(x) \rightarrow \neg \exists y\, \texttt{create}(y) \wedge (d^{=}_{1,2}(y,x) \vee d^{=}_{1,1}(y,x))$

Example 7. Messages are always sent from younger processes to older processes can be said by the formula $\forall x\, msg(x) \rightarrow d^{<}_{2,1}(x,x)$

Example 8. Every created process eventually sends a message to the "root" process. This can be said by the formula $\exists x(\forall z\,(x \leq z) \wedge \forall y\,(\texttt{create}(y) \rightarrow \exists y'\,(d^{=}_{2,1}(y,y') \wedge d^{=}_{1,2}(x,y'))))$. The formula holds in the data word *trace* (Figure 1(a)).

Example 9. This example demonstrates that our logic is powerful enough to express causal dependencies, though it is evaluated on linearizations. Two events are causally dependent can be said by the following formula $x \preceq y := (x \leq y \wedge \bigvee_{i,j\in\{1,2\}} d^{=}_{i,j}(x,y))^{*}$. We do not explicitly give this formula, but transitive closure is definable in MSO.

3 Data Pushdown automata

A data pushdown automaton is a finite state automaton equipped with a stack and a few registers. It can remember data values by either storing it in registers or by pushing it to the stack. At any state it optionally pop the topmost values in the stack to some registers (determined by a mapping ϕ). Then it can perform an event involving the data values in the registers. Then it can push some register contents to the stack, reassign the register values (by a mapping ρ), and update its state.

All registers except one are undefined in the beginning. They hold a special value $\perp$. Only the contents of those registers with a proper pid can be pushed onto the stack. Thus the stack does not contain $\perp$. Similarly the registers can be rewritten by only pids. Thus a register if ever gets to store a pid, it will never hold $\perp$ again.

The infinite set of transition labels allow a finite abstraction by writing the register name which contains the data value rather than the actual data value. Let $\mathscr{R}$ be the set of register names. The set of such abstract events is $\texttt{Events}_{\mathscr{R}}$. That is, $\texttt{Events}_{\mathscr{R}} = \{a(p_1,\ldots p_n) \mid a \in \texttt{Events}, n = arity(a)$ and p_i is a register name from the set $\mathscr{R}\}$. If the automaton executes a create action, the data value in the target register is rewritten by a "fresh" value which is higher than any of the data values used so for. Then it optionally pushes some of its register contents to the stack and updates its registers and state. We define these notions formally.

Definition 10 (data multi-pushdown automaton)**.** Let $\Bbbk \geq 0$. A $\Bbbk$-register data pushdown automaton (DPA) over $(\texttt{Events})$ is a 7-tuple $\mathscr{A} = (S, \mathscr{Z}, s_0, r_0, Z, F, \Delta)$ where S is a finite *set of states*, $\mathscr{Z}$ is a finite ranked alphabet of *stack symbols*, $s_0 \in S$ is the *initial state*, r_0 is the initial register, $Z \in \mathscr{Z}$ is the *start symbol* with $arity(Z) = 0$, and $F \subseteq S$ is the set of *final states*. Moreover, Δ is a set of *transitions* of the form $\tau = (s, A, \phi, \alpha, \texttt{upd}, \rho, s')$ where $s, s' \in S$ are states, $A \in \mathscr{Z}$, $\alpha \in \texttt{Events}_{\mathscr{R}}$ and $\texttt{upd} \in \mathscr{Z}^{*}_{\mathscr{R}}$ and $\phi : [arity(A)] \mapsto [\Bbbk]$ and $\rho : [\Bbbk] \mapsto [\Bbbk]$ are two injective partial functions.

We let $Conf_{\mathscr{A}} := S \times (\mathbb{N} \cup \{\perp\})^{\Bbbk} \times \mathbb{N} \times \mathscr{Z}^{*}_{\mathbb{N}}$ denote the set of *configurations* of $\mathscr{A}$. Configuration $\gamma = [s, \mathbf{r}, \texttt{max}, w]$ with $\mathbf{r} = (d_1, \ldots, d_{\Bbbk})$ says that the current state is s, the content of register r_i is d_i, all the

data values which have already been used are at most max, and the stack content is w where we assume that the topmost symbol is written last. If some d_i is $\perp$, then the register r_i is undefined.

Now, consider a transition $\tau = (s, A, \phi, \alpha, \mathsf{upd}, \rho, s')$. It is enabled at γ if 1) $w = w'A(d'_1, \ldots, d'_m)$, 2) $r_i \in \mathsf{pre\text{-}image}(\rho) \setminus \mathsf{image}(\phi)$, then $d_i \neq \perp$ and 3) depending on the type of α

- if $\alpha = \mathtt{create}(r_i, r_j)$, then $d_i \neq \perp$ or $i \in \mathsf{image}(\phi)$.
- if $\alpha = a(r_i, r_j, \ldots)$, then ($d_i \neq \perp$ or $i \in \mathsf{image}(\phi)$) and ($d_j \neq \perp$ or $j \in \mathsf{image}(\phi)$).

That is, for τ to be enabled at γ 1) the top stack symbol of γ should match that of the transition, 2) the register assignment should not overwrite a defined register with $\perp$, and 3) the pids executing the event must exist (or the corresponding register names must be defined).

A register assignment function $\sigma : \mathscr{R} \mapsto \mathbb{N} \cup \{\perp\}$ is said to be *suitable* for γ and τ if it is compatible with the $\mathbf{r}$ and ϕ, and moreover, if it is a create event, the target register should be assigned a value larger than max. That is:

- if $\alpha = \mathtt{create}(-, r_j)$ then $\sigma(r_j) = m$ for some $m >$ max and for $i \neq j$ if $i \in \mathsf{image}(\phi)$ then $\sigma(r_i) = d'_{\phi^{-1}(i)}$ else $\sigma(r_i) = d_i$
- otherwise, for all i if $i \in \mathsf{image}(\phi)$ then $\sigma(r_i) = d'_{\phi^{-1}(i)}$ else $\sigma(r_i) = d_i$

For every γ and τ there exists infinitely many suitable register assignment functions. If $\alpha = a(p_1, \ldots, p_n) \in \mathtt{Events}_{\mathscr{R}}$, we let $\sigma(\alpha)$ be $a(\sigma(p_1), \ldots, \sigma(p_n))$. We lift this notion to words in $\mathtt{Events}^*_{\mathscr{R}}$ as well: $\sigma(uv) = \sigma(u)\sigma(v)$.

If τ is enabled at γ and if σ is a suitable register assignment function, the automaton $\mathscr{A}$ can execute τ under σ generating $\sigma(\alpha)$. Then it moves into a new configuration $\gamma' = [s', \mathbf{r}', \mathtt{max}', w'']$ with $\mathtt{max}' = \max_i \sigma(r_i)$, $w'' = w'\sigma(\mathsf{upd})$ and $\mathbf{r}' = (\sigma(\rho^{-1}(r_1)), \ldots, \sigma(\rho^{-1}(r_\Bbbk)))$ where we set $\rho(i) = i$ if $i \notin \mathsf{image}(\rho)$. In this case we write $\gamma \overset{\sigma(\alpha)}{\Longrightarrow}_{\sigma,\tau} \gamma'$.

A configuration of the form $[s_0, (d, \perp, \ldots, \perp), d, Z]$ with $d \in \mathbb{N}$ is called *initial*, and a configuration $[s, \mathbf{r}, d, w]$ such that $s \in F$ is called *final*. A *run* of $\mathscr{A}$ on $u \in \mathtt{Events}^*_{\mathbb{N}}$ is a sequence $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\sigma_1,\tau_1} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\sigma_2,\tau_2} \ldots \overset{\alpha_n}{\Longrightarrow}_{\sigma_n,\tau_n} \gamma_n$ such that $u = \alpha_1 \cdots \alpha_n$ and γ_0 is initial. The run is *accepting* if γ_n is final. We let $L(\mathscr{A}) := \{u \in \mathtt{Events}^*_{\mathbb{N}} \mid \text{there is an accepting run of } \mathscr{A} \text{ on } u\}$ be the *language* of $\mathscr{A}$.

Example 11. A DPA for the peer-to-peer protocol (cf. Example 2) is given in Figure 2(a). It uses three registers and a stack symbol with *arity* 1. We remove this symbol in the figure for readability.

Example 12. The DPA given in Figure 2(b) accepts the distant-relative search example (cf. Example 3).

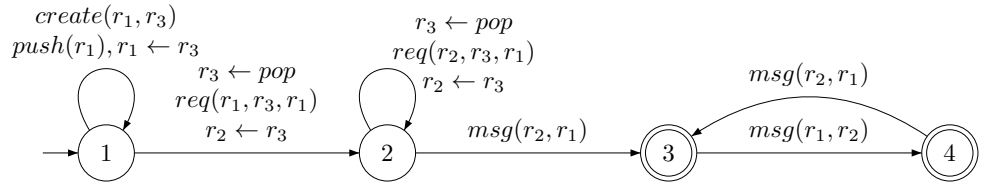
We conclude this section with two complexity results on DPAs. We have non-emptiness checking in NP and, decidable MSO model checking.

Theorem 13. *Non-emptiness checking of data pushdown automata is NP- Complete*

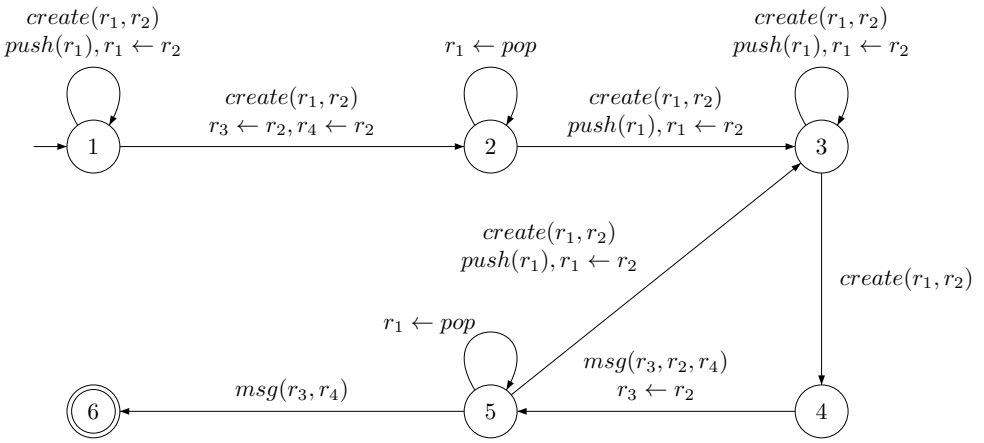
Proof. (Sketch.) The set of defined registers (ones that holds pids instead of $\perp$) is monotonously non-decreasing. Hence the NP algorithm guesses a path along with the set of defined registers, and verifies it is accepting. The lower bound is by reduction from CNF-SAT. □

Theorem 14. *Given a DPA $\mathscr{A}$ and an* MSO_d *formula ϕ, it is decidable to check whether $L(\mathscr{A}) \subseteq L(\phi)$.*

Proof. (Sketch.) The proof is by reduction to trees (or nested words) over finite alphabet. The data comparison can be recovered by classical MSO over trees (or nested words). □



(a) A DPA for peer-to-peer



(b) A DPA for distant-relative search

Figure 2: DPA examples

4 Discussions

The MSO model checking result could be extended to a DPA that runs over arbitrary data words (that is, not necessarily over `create` and `msg` alphabet). Indeed, we need to require the fresh data values to be higher than any of the previously used values. Perhaps it is also possible, instead of requiring the fresh data value to be higher, to allow guards involving data inequality comparisons of the register contents and the fresh value for the transitions.

To conclude, we have considered a special case of the Data Multipushdown automata defined in [1]. We have extended this restriction to include data comparison, while restricting the application domain to Dynamic Distributed Systems. This model is powerful enough to model several interesting examples. We retain all the results of [1], but also show a tight bound on the complexity of deciding non-emptiness for this particular class of automata.

References

[1] B. Bollig, A. Cyriac, P. Gastin and K. Narayan Kumar. Model Checking Languages of Data Words. In L. Birkedal (ed.), *FoSSaCS'12*, volume 7213 of *LNCS*, pages 391-405. Springer, 2012.

[2] B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In F. M. Ablayev and E. W. Mayr, editors, *CSR'10*, volume 6072 of *LNCS*, pages 48–59, 2010.

[3] M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.

Model Checking Flat Counter Systems
(Extended Abstract)

Amit Kumar Dhar
LIAFA, Univ. Paris Diderot
Sorbonne Paris Cité, CNRS, France
dhar@liafa.univ-paris-diderot.fr

Abstract

Reachability and model-checking problems for flat counter systems are known to be decidable, but whereas the reachability problem can be shown to be in NP, for most model checking problems the best known upper bound is made of a tower of several exponentials, when there are such bounds. Herein, we investigate and provide new results on the optimal complexity bounds for model checking problems of flat counter systems with linear-time properties using several different specification languages like temporal logic, first-order logic and automata. We also show that even extending the temporal logic with arithmetical constraints on counters preserves the bounds.
PhD advisors : Stéphane Demri (LSV) and Arnaud Sangnier (LIAFA).

1 Introduction

Counter systems are finite-state automata equipped with program variables interpreted over non-negative integers. They are used in many places like, broadcast protocols and program with pointers to quote a few. Alongwith their large scope of usability, many problems on general counter systems are known to be undecidable. But decidability of many problems can be gained by considering subclasses of counter systems. Here we consider an important and natural class of counter systems, called *flat* counter systems, i.e. whose control graph does not contain a state which occurs in more than one cycle in control graph. Many of the naturally occuring systems are inherently non-flat, for example, programs with nested loop. Though, for such systems we can under-approximate the runs to a flat system, which is useful for the existential model checking. Even though flatness is clearly a substantial restriction, it is shown in [10] that many classes of counter systems with computable Presburger-definable reachability sets are *flattable*, i.e. there exists a flat unfolding of the counter system with identical reachability sets. Decidability results on verifying safety and reachability properties on this kind of systems have been obtained in [4, 7, 2]. However, so far, such properties have been rarely considered in the framework of any formal specification language (with an exception in [3, 6]).

Here, we establish decidability results and several computational complexity characterizations of model-checking problems restricted to flat counter systems. We consider several formal specification languages like, linear temporal logic (LTL) with past, non-deterministic Büchi automata and first order logic. We show that the model checking problem is NP-Complete for specifications in LTL with past and non-deterministic Büchi automata. We also prove that model checking first order logic formulas on flat counter systems is PSPACE-Complete. The main techniques involved in obtaining these results are stuttering theorem, characterizing runs by quantifier-free Presburger formulae and small solutions of systems of equations. The detailed procedures and proofs for some of the results presented in Section 3.2 and 3.3 will be part of a forthcoming submission.

2 Definitions

Counter constraints are defined below as a subclass of Presburger formulae whose free variables are understood as counters. Such constraints are used to define guards in counter systems but also to define arithmetical constraints in formulae. Let $\mathtt{C} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ be a countably infinite set of *counters* (variables interpreted over non-negative integers) and $\mathrm{AT} = \{p_1, p_2, \ldots\}$ be a countable infinite set of propositional variables (abstract properties about program points). We write $\mathtt{C}_n$ to denote the restriction of $\mathtt{C}$ to $\{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$.

Definition 1 (Guards)**.** The set $\mathtt{G}(\mathtt{C}_n)$ of *guards* (arithmetical constraints on counters in $\mathtt{C}_n$) is defined inductively as follows:

$$\begin{array}{lcl} \mathtt{t} & ::= & a.\mathtt{x} \mid \mathtt{t}+\mathtt{t} \\ \mathtt{g} & ::= & \mathtt{t} \sim b \mid \mathtt{g} \wedge \mathtt{g} \mid \mathtt{g} \vee \mathtt{g} \end{array}$$

where $\mathtt{x} \in \mathtt{C}_n$, $a \in \mathbb{Z}$, $b \in \mathbb{N}$ and $\sim \in \{=, \leq, \geq, <, >\}$.

Note that such guards are closed under negations (but negation is not a logical connective) and the truth constants $\top$ and $\bot$ can be easily defined too. Given $\mathtt{g} \in \mathtt{G}(\mathtt{C}_n)$ and a vector $\mathbf{v} \in \mathbb{N}^n$, we say that $\mathbf{v}$ satisfies $\mathtt{g}$, written $\mathbf{v} \models \mathtt{g}$, if the formula obtained by replacing each $\mathtt{x}_i$ by $\vec{v}[i]$ holds. For example $\mathtt{x}_1 \geq 0 \wedge \mathtt{x}_2 \leq 7$ is a guard from Figure 1 and the vector $(3,-1)$ satisfies it.

2.1 Flat Counter Systems

Definition 2 (Counter system)**.** For a natural number $n \geq 1$, a n-dim counter system (shortly a counter system) S is a tuple $\langle Q, \mathtt{C}_n, \Delta, \mathbf{l} \rangle$ where:

- Q is a finite set of *control states*.
- $\Delta \subseteq Q \times \mathtt{G}(\mathtt{C}_n) \times \mathbb{Z}^n \times Q$ is a finite set of edges labeled by guards and updates of the counter values (*transitions*).
- $\mathbf{l} : Q \to 2^{\mathrm{AT}}$ is a *labelling function*.

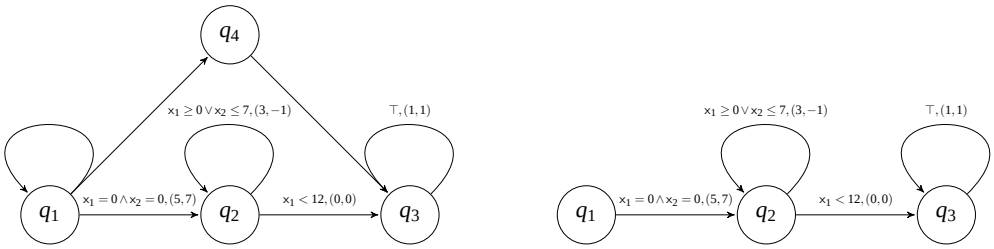
For $\delta = \langle q, \mathtt{g}, \mathbf{u}, q' \rangle$ in Δ, we denote $\mathtt{g}$ and $\mathbf{u}$ as $\mathit{guard}(\delta)$ and $\mathit{update}(\delta)$ respectively.

We denote the configuration set of S as $C = Q \times \mathbb{N}^n$. Given an initial configuration $c_0 \in Q \times \mathbb{N}^n$, a *run* ρ starting from c_0 in S is an infinite sequence of configurations such that it describes a path in S and is denoted as: $\rho := c_0 \xrightarrow{\delta_0} \cdots \xrightarrow{\delta_{m-1}} c_m \xrightarrow{\delta_m} \cdots$, where $c_i = \langle q_i, \mathbf{v}_i \rangle \in Q \times \mathbb{N}^n$, $\delta_i \in \Delta$, $\mathbf{v}_i \models \mathit{guard}(\delta_i)$ and $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathit{update}(\delta)$ for all $i \in \mathbb{N}$. For example, a run in the counter system in Figure 1, could be

$$\langle q_1, (0,0) \rangle \to \langle q_2, (5,7) \rangle \to \langle q_2, (8,6) \rangle \to \langle q_2, (11,5) \rangle \to \langle q_3, (11,5) \rangle \to \langle q_3, (12,6) \rangle \cdots$$

We say that a counter system is *flat* if every node in the underlying graph belongs to at most one simple cycle (a cycle being simple if no edge is repeated twice in it) [4]. In a flat counter system, simple cycles can be organized as a DAG where two simple cycles are in the relation whenever there is path between a node of the first cycle and a node of the second cycle. We denote by $\mathtt{CFS}$ the class of flat counter systems. A *flat Kripke structure* can be thought of as a flat counter system without any counter and thus do not have any guards and updates on transitions. The class of flat Kripke structures are denoted by $\mathtt{KFS}$. The counter system shown in Figure 1(a) is a flat counter system.

A path schema, P in a system is an ω-regular expression of the form $p_1 l_1^+ p_2 l_2^+ \cdots p_k l_k^\omega$, where each p_i is a sequence of consecutive edges called a path segment in the given system and each l_i is called a loop segment is a path segment which describes a cycle in the given system. A *minimal* path schema



(a) Flat counter system (b) Path schema

Figure 1: Counter system and one of its path schema

is a path schema where no edge is repeated strictly more than twice. Note that there are exponentially many minimal path schemas in a flat counter system. The language of a path schema represents a set of runs which visit the loops in P in order at least once and the last loop infinitely many times. For a run ρ, belonging to the language of path schema, we also say that ρ respects P. Figure 1(b) shows a path schema in the counter system. Note that ρ in the example above respects the path schema in Figure 1(b).

2.2 Specification Languages

Models of all the logics presented here are essentially abstractions of runs from counter systems. Since a run in a flat counter system is an infinite sequence of configurations we can represent the runs as ω-sequences $\sigma : \mathbb{N} \to 2^{\mathrm{AT}} \times \mathbb{N}^{\mathtt{C}}$, which we take as model for all the specifications defined here.

Formulae of PLTL$[\mathtt{C}]$ are defined as usual with exception of adding guards as atomic propositions. The formulas consist of $\phi ::= p \mid \mathtt{g} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathtt{X}\phi \mid \phi \mathtt{U} \phi \mid \mathtt{X}^{-1}\phi \mid \phi \mathtt{S} \phi$ where $p \in \mathrm{AT}$ and $\mathtt{g} \in \mathtt{G}(\mathtt{C}_n)$ for some n. We also denote by PLTL$[\emptyset]$ the $PLTL[\mathtt{C}]$ formulas where no guards appear in the formula. Given a model σ and a position $i \in \mathbb{N}$, the satisfaction relation $\models$ for PLTL$[\mathtt{C}]$ is defined as usual for the Boolean combinations and temporal operators. For example:

- $\sigma, i \models \mathtt{g} \overset{\mathsf{def}}{\Leftrightarrow} \mathbf{v}_i \models \mathtt{g}$ where $\mathbf{v}_i[j] \overset{\mathsf{def}}{=} \pi_2(\sigma(i))(\mathtt{x}_j)$,
- $\sigma, i \models \mathtt{X}\phi \overset{\mathsf{def}}{\Leftrightarrow} \sigma, i+1 \models \phi$,
- $\sigma, i \models \phi_1 \mathtt{S} \phi_2 \overset{\mathsf{def}}{\Leftrightarrow} \sigma, j \models \phi_2$ for some $0 \leq j \leq i$ s.t. $\sigma, k \models \phi_1, \forall j < k \leq i$.

For example a PLTL$[\mathtt{C}]$ formula could be $(\mathtt{x}_1 < 10)\mathtt{U}(a \wedge (\mathtt{x}_2 \geq 2) \wedge \mathtt{X}b)$ which is satisfied by the example run presented above with the labelling function mapping q_2 to $\{a\}$ and q_3 to $\{b\}$.

We define a Büchi automaton, $\mathscr{B}$ as usual over a finite alphabet set Σ consisting of all the subsets of AT. Thus, the language accepted by $\mathscr{B}$ is a subset of $(2^{\mathrm{AT}})^{\omega}$. Hence, we say $\sigma \models \mathscr{B}$ iff $\mathsf{proj}_1(\sigma)$, projection of the first component of every configuration in σ, belongs to $\mathscr{L}(\mathscr{B})$.

For defining First Order logic, we assume a countably infinite set of variables denoted by $\mathtt{z}_1, \mathtt{z}_2, \ldots$. We also assume a set of propositional symbols denoted as AT. A formula in *First Order* ($FO[\mathtt{C}]$) logic with counters is constructed as: $\phi ::= P_a(\mathtt{z}) \mid \mathtt{g}(\mathtt{z}) \mid \mathtt{z}_1 < \mathtt{z}_2 \mid \mathtt{z}_1 = \mathtt{z}_2 \mid \neg\phi \mid \phi \wedge \phi' \mid \exists \mathtt{z} \phi(\mathtt{z})$, where $a \in AT$. The variables of a $FO[\mathtt{C}]$ formula are interpreted over the positions in the model. The satisfaction relation of $FO[\mathtt{C}]$ formulas are defined as usual. We write $FO[\emptyset]$ to denote the restriction of $FO[\mathtt{C}]$ without

arithmatical constraints $g(z)$. For example $\exists z_1 z_2 z_3 . P_a(z_1) \wedge P_a(z_2) \wedge P_a(z_3) \wedge \forall z_4 P_a(z_4) \wedge (z_4 = z_1 \vee z_4 = z_2 \vee z_4 = z_3)$ is $FO[\emptyset]$ sentence which is satisfied by the example run above with the labelling function mapping q_2 to $\{a\}$ and q_3 to $\{b\}$.

It is interesting to note that there exists a standard logspace translation from $PLTL[\mathtt{C}]$ formula to a $FO[\mathtt{C}]$ formula, while preserving the semantics. In fact, as presented here, $FO[\mathtt{C}]$ and $PLTL[\mathtt{C}]$ have the same expressive power, thanks to Kamp's theorem. On the other hand $PLTL[\mathtt{C}]$ can not be translated to Büchi automata because of the guards in the logic. However, it is possible to have an exponential translation from $PLTL[\emptyset]$ formula to Büchi automata.

2.3 Problem Definition

Effectively, a run in a flat counter system is an infinite sequence of configurations from the set $2^{\mathrm{AT}} \times \mathbb{N}^{\mathtt{C}}$. Thus, we can easily see a run ρ from a flat counter system as a model for logics $\sigma : \mathbb{N} \to 2^{\mathrm{AT}} \times \mathbb{N}^{\mathtt{C}}$ which returns the configuration of the run for every position in the ρ. Given a specification language L and a class $\mathscr{C}$ of counter systems, we write $\mathrm{MC}(\mathrm{L}, \mathscr{C})$ to denote the existential model checking problem: given $S \in \mathscr{C}$, a configuration c_0 and $\phi \in \mathrm{L}$, does there exist ρ starting from c_0 such that $\rho, 0 \models \phi$? In that case, we write $S, c_0 \models \phi$. It is known that for the full class of counter systems, even the reachability problem is undecidable, see e.g. [11]. Some restrictions, such as flatness, can lead to decidability as shown in [6] but the decision procedure there involves an exponential reduction to Presburger Arithmetic, whence the high complexity.

3 Results

We will now state the results related to the problems $\mathrm{MC}(PLTL[\mathtt{C}], \mathtt{CFS})$, $\mathrm{MC}(FO[\mathtt{C}], \mathtt{CFS})$. We will first list the basic steps followed by the algorithms. In effect, though the steps followed by the algorithms are the same, the techniques used and the bound obtained for the steps are different in each case and this difference leads to different complexity characterization. Given a flat counter system S and a formula in the respective logic ϕ, basically the following steps are followed:

1. Select a minimal path schema P from S.
   - There are exponentially many path schemas in a flat counter system and their size is bounded by $\mathrm{size}(S)$.
2. Guess an "equivalent" path schema P' to P such that P' contains no disjunction in guards and the states are labelled with counter constraints that are satisfied at the specific state.
   - From [5], we know that, there are polynomially many path schemas equivalent to P.
3. Guess a quantifier-free Presburger formula ψ characterizing which loops to be taken a fixed number of times and which ones can be taken unbounded number of times. Check that the solutions to ψ give runs that satisfy the given specification ϕ when the counter behaviours in P' are ignored.
   - This can be done by establishing "stuttering theorem" (similar to the pumping lemma for finite automaton) for each specification.
4. Build constraint system $\mathscr{E}$ that characterizes precisely the number of times each loop can be taken in a valid run (characteristics of the counters are taken into consideration).
   - From [5], we know that, we can build $\mathscr{E}$ in polynomial time.
5. Guess and check a polynomial size solution for $\mathscr{E} \wedge \psi$.

- From [1], we know that, there exists a polynomial size solution if it is satisfiable.

Note that Step 2 and 4 does not depend on the specification language and hence is same for different logics. On the other hand Step 3 depends only on the specification and the structure of the system. Thus, the stuttering theorem and procedure for checking satisfaction of formula are different for various logics.

3.1 Linear Temporal Logic with Past

The "stuttering theorem" for $PLTL[\emptyset]$ formulas is proved in [5]. It gives a bound on the number of times each loop in a path schema from flat Kripke structure is taken, depending on the temporal depth of the formula. The temporal depth of a $PLTL[\emptyset]$ formula is the maximum number of nesting of the temporal operators in the formula. The bound given by the theorem is $2.td(\phi') + 5$ where $td(\phi)$ is the temporal depth of ϕ. Clearly, the bound is polynomial in size(ϕ). Thus, we can unfold loop i of P' $\vec{y}[i]$ times, giving us an *ultimately periodic path* of polynomial size. Model checking ϕ' on such a path can be done in polynomial time by the procedure in [9]. Note that every step of the algorithm can be completed in polynomial time and with guesses of polynomial size. Thus, it gives us an NP procedure for the problem.

The NP-hardness of the problem follows from a reduction from Boolean satisfiability problem, where the truth value of the variables are encoded by the number of times a fixed loop is taken.

Theorem 3. MC$(PLTL[\mathtt{C}], \mathtt{CFS})$ *is* NP*-Complete.*

3.2 First-Order Logic

The "stuttering theorem" for $FO[\emptyset]$ formulas is proved using Ehrenfeucht-Fraïssé games (EF Games). It gives a bound on the number of times each loop in a path schema from flat Kripke structure is taken, depending on the quantifier height of the formula. The quantifier height of a $FO[\emptyset]$ formula is the maximum number of nesting of the $\exists$ operator in the formula. The bound given by the theorem is $2^{qh(\phi')}$ where $qh(\phi')$ is the quantifier height of ϕ'. Clearly, the bound is exponential in size(ϕ'). Thus, we can not directly construct an ultimately periodic path from P' and $\vec{y}$ as the path could be exponential in length. Instead we work with the succinct representation of this exponential path as $\langle P', \vec{y} \rangle$. Recall that, even though $\vec{y}$ may contain exponential value, the number of bits required to represent $\vec{y}$ is polynomial due to binary encoding. Now, since the variables in ϕ' are interpreted over the positions in run which can be at most $2^{qh(\phi')} \times$ size(P'), this can also be stored using polynomially many bits. And evaluating ϕ' on such a representation can be done using arithmetic over polynomially many bits. Note that every step of the algorithm can be completed in polynomial space and with guesses of polynomial size. Thus, it gives us a PSPACE procedure for the problem.

The PSPACE-hardness of the problem follows from a reduction from quantified boolean formula on finite structures.

Theorem 4. MC$(FO[\mathtt{C}], \mathtt{CFS})$ *is* PSPACE*-Complete.*

3.3 Büchi Automata

For MC$(BA, \mathtt{CFS})$ we employ a different procedure. We use the fact that for a non-deterministic finite automaton, there exists a existential Presburger formula of polynomial size which characterizes the Parikh image of its language [12]. We also use the fact that an accepting run in a Büchi automaton, can be split non-deterministically into two parts, each part recognised by a non-deterministic finite state automaton. The procedure involves constructing a product automata for each part with the given path schema and check for satisfiability of the formula characterizing the Parikh images.

Theorem 5. $\mathrm{MC}(BA, \mathtt{CFS})$ *is* NP*-Complete.*

From the above result we can conclude that if we define liner μ-calculus in a way similar to $PLTL[\mathtt{C}]$, we can obtain that the model checking problem for such logic is decidable. This can be done by performing the translation from linear μ-calculus to Büchi automata and then applying the procedure for $\mathrm{MC}(BA, \mathtt{CFS})$.

4 Conclusion

Here, we have investigated the complexity of the model-checking problem of various kinds of specifications for linear time properties over flat counter systems and its subclasses. The main results showing NP-completeness of the problem $\mathrm{MC}(\mathrm{PLTL}[\mathtt{C}], \mathtt{CFS})$ improve the upper bound of previously known results and also improves the understanding of flat counter systems by showing the PSPACE-completeness of $\mathrm{MC}(FO[\mathtt{C}], \mathtt{CFS})$ which was previously not known. Furthermore the results extend the result from [8], by including past time operator in the logic and showing that it still has the same complexity. The results are represented in the following table.

System	PLTL$[\emptyset]$	PLTL$[\mathtt{C}]$	$FO[\emptyset]$	$FO[\mathtt{C}]$	BA
KFS	NP-complete	-	PSPACE-complete	-	NP-complete
CFS	NP-complete	NP-complete	PSPACE-complete	PSPACE-complete	NP-complete

References

[1] I. Borosh and L. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *American Mathematical Society*, 55:299–304, 1976.

[2] M. Bozga, R. Iosif, and F. Konecný. Fast acceleration of ultimately periodic relations. In *CAV'10*, volume 6174 of *LNCS*, pages 227–242. Springer, 2009.

[3] H Comon and V. Cortier. Flatness is not a weakness. In *CSL'00*, volume 1862 of *LNCS*, pages 262–276. Springer, 2000.

[4] H. Comon and Y. Jurski. Multiple counter automata, safety analysis and PA. In *CAV'98*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.

[5] S. Demri, A.K. Dhar, and A. Sangnier. Taming Past LTL and Flat Counter Systems. In *IJCAR'12*, volume 7364 of *LNAI*, pages 179–193. Springer, 2012.

[6] S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Model-checking CTL* over flat Presburger counter systems. *Journal of Applied Non-Classical Logic*, 20(4):313–344, 2010.

[7] A. Finkel and J. Leroux. How to compose Presburger accelerations: Applications to broadcast protocols. In *FST&TCS'02*, volume 2256 of *LNCS*, pages 145–156. Springer, 2002.

[8] L. Kuhtz and B. Finkbeiner. Weak Kripke structures and LTL. In *CONCUR'11*, volume 6901 of *LNCS*, pages 419–433. Springer, 2011.

[9] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *LICS'02*, pages 383–392. IEEE, 2002.

[10] J. Leroux and G. Sutre. Flat counter systems are everywhere! In *ATVA'05*, volume 3707 of *LNCS*, pages 489–503. Springer, 2005.

[11] M. Minsky. *Computation, Finite and Infinite Machines*. Prentice Hall, 1967.

[12] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. On the complexity of equational horn clauses. In *CADE*, pages 337–352, 2005.

Inferring Biological Regulatory Networks from Process Hitting models

Maxime Folschette

LUNAM Université, École Centrale de Nantes, IRCCyN UMR CNRS 6597
(Institut de Recherche en Communications et Cybernétique de Nantes)
1 rue de la Noë – B.P. 92101 – 44321 Nantes Cedex 3, France.

National Institute of Informatics,
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan.

Maxime.Folschette@irccyn.ec-nantes.fr

Joint work with: Loïc Paulevé, Katsumi Inoue, Morgan Magnin, Olivier Roux

Abstract

The Process Hitting (PH) is a recently introduced framework to model concurrent processes. It is notably suitable to model Biological Regulatory Networks (BRNs) with partial knowledge of cooperations by defining the most permissive dynamics. On the other hand, the qualitative modeling of BRNs has been widely addressed using René Thomas' formalism. Given a PH model of a BRN, we first tackle the inference of the underlying Interaction Graph between components. Then the inference of corresponding Thomas' models is provided by inferring some parameters and enumerating the compatible parametrizations.

1 Introduction

As regulatory phenomena play a crucial role in biological systems, they need to be studied accurately. Biological Regulatory Networks (BRNs) consist in sets of either positive or negative mutual effects between the components. Besides continuous models of physicists, often designed through systems of ordinary differential equations, a discrete modeling approach was initiated by René Thomas in 1973 [16] allowing the representation of the different levels of a component, such as concentration or expression levels, as integer values. Nevertheless, these dynamics can be precisely established only with regard to some kind of "focal points", related to as Thomas' parameters, indicating the evolutionary tendency of each component. This modeling has motivated numerous works (e.g., [12, 9, 15, 1]), and other approaches related to our work, which rely on temporal logic [7] and constraint programming [4, 5], aim at determining models consistent with partial data on the regulatory structure and dynamics. While the formal checking of dynamical properties is often limited to small networks because of the state graph explosion, the main drawback of this framework is the difficulty to specify Thomas' parameters, especially for large networks.

In order to address the formal checking of dynamical properties within very large BRNs, we recently introduced in [10] a new formalism, named the *"Process Hitting"* (PH), to model concurrent systems having components with a few qualitative levels. A PH describes, in an atomic manner, the possible evolutions of a "process" (representing one component at one level) triggered by the hit of at most one other "process" in the system. This particular structure makes the formal analysis of BRNs with hundreds of components tractable [11]. PH is suitable, according to the precision of this information, to model BRNs with different levels of abstraction by capturing the most general dynamics.

In this work[1], we show that starting from one PH model, it is possible to find the underlying interactions, then the underlying Thomas' parameters. It relies on an exhaustive search of the interactions between components of the PH model, and an enumeration of the (possibly large) nesting set of valid parameters, so that the resulting dynamics are ensured to respect the PH dynamics, i.e. no spurious transitions are made possible.

The first benefit of our approach is that it makes possible the construction refining of BRNs with a partial and progressively brought knowledge in PH, while being able to export such models in the Thomas' framework. Our second contribution is to enhance the knowledge of the formal links between both modelings. The method can be applied to large BRNs (up to 40 components).

2 Frameworks

2.1 The Process Hitting framework

A Process Hitting (PH) (Def. 1) gathers a finite number of concurrent *processes* grouped into a finite set of *sorts*. A sort stands for a component of the system while a process, which belongs to a unique sort, stands for one of its expression levels. A process is noted a_i where a is the sort and i is the process identifier within the sort a. At any time, exactly one process of each sort is present; a *state* of the PH corresponds to such a set of processes.

The concurrent interactions between processes are defined by a set of *actions*. Actions describe the replacement of a process by another of the same sort conditioned by the presence of at most one other process in the current state. An action is denoted by $a_i \to b_j \nearrow b_k$, which is read as "a_i *hits* b_j to make it bounce to b_k", where a_i, b_j, b_k are processes of sorts a and b, called respectively *hitter*, *target* and *bounce* of the action.

Definition 1 (Process Hitting)**.** A *Process Hitting* is a triple $(\Sigma, L, \mathscr{H})$:

- $\Sigma = \{a, b, \dots\}$ is the finite set of *sorts*;
- $L = \prod_{a\in\Sigma} L_a$ is the set of states with $L_a = \{a_0, \dots, a_{l_a}\}$ the finite set of *processes* of sort $a \in \Sigma$ and l_a a positive integer, with $a \neq b \Rightarrow L_a \cap L_b = \emptyset$;
- $\mathscr{H} = \{a_i \to b_j \nearrow b_k \in L_a \times L_b \times L_b \mid (a,b) \in \Sigma^2 \wedge b_j \neq b_k \wedge a = b \Rightarrow a_i = b_j\}$ is the finite set of *actions*.

Given a state $s \in L$, the process of sort $a \in \Sigma$ present in s is denoted by $s[a]$. An action $h = a_i \to b_j \nearrow b_k \in \mathscr{H}$ is *playable* in $s \in L$ if and only if $s[a] = a_i$ and $s[b] = b_j$. In such a case, $(s \cdot h)$ stands for the state resulting from the play of the action h in s, with $(s \cdot h)[b] = b_k$ and $\forall c \in \Sigma, c \neq b, (s \cdot h)[c] = s[c]$.

Modeling cooperation. As described in [10], the cooperation between processes to make another process bounce can be expressed in PH by building a *cooperative sort*. Fig. 1 shows an example of a cooperative sort bc between sorts b and c, defined with 4 processes (one for each sub-state of the presence of processes b_1 and c_1). For the sake of clarity, processes of bc are indexed using the sub-state they represent. Hence, bc_{01} represents the sub-state $\langle b_0, c_1 \rangle$, and so on. Each process of sort b and c hit bc to make it bounce to the process reflecting the status of the sorts b and c (e.g., $b_1 \to bc_{00} \nearrow bc_{10}$ and $b_1 \to bc_{01} \nearrow bc_{11}$). Then, to represent the cooperation between processes b_1 and c_1, the process bc_{11} hits a_1 to make it bounce to a_2 instead of independent hits from b_1 and c_1. The same cooperative sort is used to make b_0 and c_0 cooperate to hit a_1 and make it bounce to a_0.

[1] The formal details of our method are presented in [6].

Example. Fig. 1 represents a PH $(\Sigma, L, \mathscr{H})$ with especially: $\Sigma = \{a, b, c, bc\}$, $L_a = \{a_0, a_1, a_2\}$, $L_b = \{b_0, b_1\}$, $L_c = \{c_0, c_1\}$ and $L_{bc} = \{bc_{00}, bc_{01}, bc_{10}, bc_{11}\}$. This example models a BRN where the component a has three qualitative levels, components b and c are Boolean and bc is a cooperative sort. In this BRN, a inhibits b at level 2 while b and c activate a with independent actions (e.g. $b_0 \to a_2 \not\uparrow a_1$) or through the cooperative sort bc (e.g. $bc_{11} \to a_1 \uparrow a_2$). Indeed, the reachability of a_2 and a_0 is conditioned by a cooperation of b and c, as explained above.

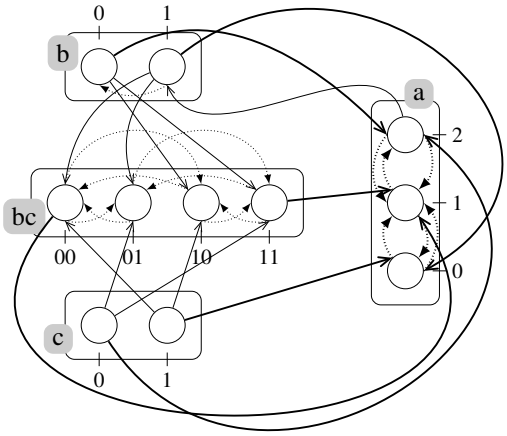


Figure 1: A PH example with four sorts: three components (a, b and c) and a cooperative sort (bc). Actions targeting processes of a are in thick lines.

2.2 Thomas' modeling

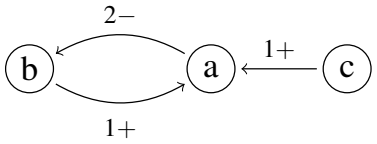
Thomas' formalism, here inspired by [13, 3], lies on two complementary descriptions of the system. First, the *Interaction Graph* (IG) models the structure of the system by defining the components' mutual influences. Its nodes represent components, while its edges labeled with a threshold stand for either positive or negative interactions (Def. 2); l_a denotes the maximum level of a component a.

Definition 2 (Interaction Graph)**.** An *Interaction Graph* (IG) is a triple (Γ, E_+, E_-) where Γ is a finite number of *components*, and E_+ (resp. E_-) $\subset \{a \xrightarrow{t} b \mid a, b \in \Gamma \wedge t \in [1; l_a]\}$ is the set of positive (resp. negative) *regulations* between two nodes, labeled with a *threshold*. A regulation from a to b is unique.

For an interaction of the IG to take place, the expression level of its head component has to be higher than its threshold; otherwise, the opposite influence is expressed. For any component $a \in \Gamma$, $\Gamma^{-1}(a) = \{b \in \Gamma \mid \exists b \xrightarrow{t} a \in E_+ \cup E_-\}$ is the set of its regulators. A *state* s of an IG (Γ, E_+, E_-) is an element in $\prod_{a \in \Gamma}[0; l_a]$ and $s[a]$ refers to the level of component a in s.

The specificity of Thomas' approach lies in the use of discrete *parameters* to represent focal level intervals (Def. 3).

Definition 3 (Discrete parameter $K_{x,A,B}$ and Parametrization K)**.** Let $x \in \Gamma$ be a given component and A (resp. B) $\subset \Gamma^{-1}(x)$ a set of its *activators* (resp. *inhibitors*), such that $A \cup B = \Gamma^{-1}(x)$ and $A \cap B = \emptyset$. The discrete *parameter* $K_{x,A,B} = [i; j]$ is a non-empty interval so that $0 \leq i \leq j \leq l_x$. With regard to the dynamics, x will tend towards $K_{x,A,B}$ in the states where its activators (resp. inhibitors) are the regulators in set A (resp. B). The complete map $K = (K_{x,A,B})_{x,A,B}$ of discrete parameters for an IG is called a *parametrization* of this IG.



$$K_{a,\{b,c\},\emptyset} = [2;2] \qquad K_{b,\{a\},\emptyset} = [0;1]$$
$$K_{a,\{b\},\{c\}} = [1;1] \qquad K_{b,\emptyset,\{a\}} = [0;0]$$
$$K_{a,\{c\},\{b\}} = [1;1]$$
$$K_{a,\emptyset,\{b,c\}} = [0;0] \qquad K_{c,\emptyset,\emptyset} = [0;1]$$

Figure 2: (left) IG example. Regulations are represented by the edges labeled with their sign and threshold. For instance, the edge from b to a is labeled "1+", which stands for: $b \xrightarrow{1} a \in E_+$. (right) Example parametrization of the left IG.

At last, dynamics are defined in BRN in a unitary and asynchronous way: from a given state s, a transition to another state s' is possible provided that only one component a will evolve of exactly one level towards $K_{a,A,B}$, where A (resp. B) is the set of activators (resp. inhibitors) of a in s.

Example. Fig. 2(left) represents an Interaction Graph (Γ, E_+, E_-) with $\Gamma = \{a,b,c\}$, $E_+ = \{b \xrightarrow{1} a, c \xrightarrow{1} a\}$ and $E_- = \{a \xrightarrow{2} b\}$; hence $\Gamma^{-1}(a) = \{b,c\}$. Fig. 2(right) gives a possible parametrization on this IG. In this BRN, the following transitions are possible: $\langle a_0,b_1,c_1\rangle \to \langle a_1,b_1,c_1\rangle \to \langle a_2,b_1,c_1\rangle \to \langle a_2,b_0,c_1\rangle \to \langle a_1,b_0,c_1\rangle$, where a_i is the component a at level i.

3 Interaction Graph Inference

In order to infer a complete BRN, one has to find the Interaction Graph (IG) first, as some constraints on the parametrization rely on it. Inferring the IG is an abstraction step which consists, from atomistic actions of a PH, in determining the global influence of every component on each of its successors.

This step assumes that the studied PH defines two types of sorts: the sorts corresponding to BRN components, which will appear in the IG, and the cooperative sorts. The identification of these two sets of sorts relies on the observation of their possible behavior, which in both cases observe some rules.

IG Inference Inferring global influences of a predecessor b on a component a requires to find "local influences" from this predecessor first, by considering a given state and changing only the active process of b. The aim is to compare the set of processes towards which the component a will evolve, for each active process of b, leaving the active process of all the other sorts unchanged. Indeed, if after increasing the level of b (i.e. activating a higher process of b) we notice that a tends to reach a higher (resp. lower) level, we can then deduce that b activates (resp. inhibits) a in this selected state. Of course, only predecessors of a have to be considered.

This has to be observed on every possible state in order to infer a local influence. Indeed, if all local influences of b on a are the same (activations or inhibitions) we can deduce that the global influence of b on a is also the same, and the related threshold is the lowest level of b for which we can observe such an influence. An unsigned edge with no threshold is inferred if two different local influences are found, or in particular cases (when a behavior cannot be represented as a BRN).

Example. Consider, in the PH of Fig. 1, the sub-state $\langle b_0,c_0,bc_{00}\rangle$ of predecessors of a. In this sub-state, a can be hit by the following actions: $\{b_0 \to a_2 \not\Uparrow a_1, c_0 \to a_2 \not\Uparrow a_1, bc_{00} \to a_1 \not\Uparrow a_0\}$. Thus, if a evolves, it will eventually reach process a_0. But if a higher process of b is activated, that is, b_1

instead of b_0, thus considering the sub-state $\langle b_1, c_0, bc_{10}\rangle$, then a can be hit by the two following actions: $\{b_1 \to a_0 \Rsh a_1, c_0 \to a_2 \Rsh a_1\}$, and will eventually reach process a_1.

Therefore, in this sub-state of predecessors of a, b locally activates a. Furthermore, if this analysis is carried for all possible sub-states of predecessors of a, only local activations are found, thus giving: $b \xrightarrow{1} a \in E_+$. After applying this method to all pairs of influence, the IG given in Fig. 2 is inferred.

4 Parametrization inference

Given the IG inferred from a PH as presented in the previous section, one can find the discrete parameters that model the behavior of the studied PH using the method presented in the following. As some parameters may remain undetermined, another step allows to enumerate all parametrizations compatible with the inferred parameters.

Independent parameters inference This subsection presents some results related to the inference of independent discrete parameters from a given PH, equivalent to those presented in [10]. We suppose in the following that the considered PH is well-formed for parameters inference, i.e. its inferred IG does not contain any unsigned edge, and in each sort, all processes activating (resp. inhibiting) another component share the same behavior. Let $K_{a,A,B}$ be the parameter we want to infer for a given component $a \in \Gamma$, and $A \subset \Gamma^{-1}(a)$ (resp. $B \subset \Gamma^{-1}(a)$) a set of its activators (resp. inhibitors). This inference, as for the IG inference, relies on the search of focal processes of the component for the given configuration of its regulators.

For each sort $b \in \Gamma^{-1}(a)$, we define a context that contains all processes of b activating (resp. inhibiting) a if $b \in A$ (resp. B). From all contexts of all predecessors of a, we create a global context that represents the configuration A, B (including the cooperative sorts involved). The parameter $K_{a,A,B}$ specifies towards which values a eventually evolves as long as this context holds, which is precisely given by the set of focal processes.

Example. Consider the PH of Fig. 1, from which the IG of Fig. 2 is inferred. Inferring the parameter $K_{a,\{b,c\},\emptyset}$ requires to understand the behavior of a in the sub-state $\langle b_1, c_1, bc_{11}\rangle$. In this sub-state, a tends to eventually reach process a_2; thus, we can deduce the parameter: $K_{a,\{b,c\},\emptyset} = [2;2]$. Inferring all parameters leads to the complete parametrization given in Fig. 2.

Admissible parametrizations enumeration The previous inference step may leave several parameters undetermined, due to missing cooperations or behaviors impossible to represent in a BRN. If it is not possible to change the PH model in order to remove these inconclusive cases, one can perform a last step to enumerate all valid values for each parameter that could not be inferred given the above results. We consider that a parameter is valid if any transition it involves in the resulting BRN is allowed by the studied PH by actions that represent this behavior. We also add some biological constraints on the whole parametrizations, given in [3]. These constraints lead to a family of admissible parametrizations which we can enumerate and are ensured to observe a coherent behavior that is included in the original PH.

Answer Set Programming (ASP) [2] turns out to be effective for the enumerative searches developed in this paper, as it efficiently tackles the inherent complexity of the models we use, thus allowing an efficient execution of the formal tools developed. Furthermore, ASP finds a particularly interesting application in the research of admissible parametrizations regarding the properties presented above, as this enumeration can be naturally formulated by using of aggregates and constraints.

5 Implementation

The inference method described in this paper has been implemented as a tool named `ph2thomas`, as part of PINT[2], which gathers PH related tools. Our implementation mainly consists of ASP programs that are solved using Clingo[3].

In the previous sections, we illustrate our results on a toy example considered as a very small network. But our approach can also successfully handle large PH models of BRNs found in the literature such as an ERBB receptor-regulated G1/S transition model from [14] which contains 20 components, and a T-cells receptor model from [8] which contains 40 components[4]. For each model, IG and parameters inferences are performed together in less than a second on a standard desktop computer.

6 Conclusion

This work establishes the abstraction relationship between PH, which is more abstract and allows incomplete knowledge on cooperations, and Thomas' approach for qualitative BRN modeling. This motivates the concretization of PH models into a set of compatible Thomas' models in order to benefit of the complementary advantages of these two formal frameworks and extract some global information about the influences between components.

As an extension of the present work, we plan to explore new semantics of BRNs to be able to tackle influences currently represented by unsigned edges.

Acknowledgment. This work was partially supported by the Fondation Centrale Initiatives.

References

[1] Jamil Ahmad, Olivier Roux, Gilles Bernot, Jean-Paul Comet, and Adrien Richard. Analysing formal models of genetic regulatory networks with delays. *International Journal of Bioinformatics Research and Applications (IJBRA)*, 4(2), 2008.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[3] Gilles Bernot, Franck Cassez, Jean-Paul Comet, Franck Delaplace, Céline Müller, and Olivier Roux. Semantics of biological regulatory networks. *Electronic Notes in Theoretical Computer Science*, 180(3):3 – 14, 2007.

[4] Fabien Corblin, Eric Fanchon, and Laurent Trilling. Applications of a formal approach to decipher discrete genetic networks. *BMC Bioinformatics*, 11(1):385, 2010.

[5] Fabien Corblin, Eric Fanchon, Laurent Trilling, Claudine Chaouiya, and Denis Thieffry. Automatic inference of regulatory and dynamical properties from incomplete gene interaction and expression data. In *IPCAT*, volume 7223 of *LNCS*, pages 25–30. Springer, 2012.

[6] Maxime Folschette, Loïc Paulevé, Katsumi Inoue, Morgan Magnin, and Olivier Roux. Concretizing the process hitting into biological regulatory networks. In David Gilbert and Monika Heiner, editors, *Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 166–186. Springer Berlin Heidelberg, 2012.

[2] Available at `http://process.hitting.free.fr`

[3] Available at `http://potassco.sourceforge.net`

[4] Both models are available as examples distributed with PINT.

[7] Z. Khalis, J.-P. Comet, A. Richard, and G. Bernot. The SMBioNet method for discovering models of gene regulatory networks. *Genes, Genomes and Genomics*, 3(special issue 1):15–22, 2009.

[8] Steffen Klamt, Julio Saez-Rodriguez, Jonathan Lindquist, Luca Simeoni, and Ernst Gilles. A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinformatics*, 7(1):56, 2006.

[9] Aurélien Naldi, Elisabeth Remy, Denis Thieffry, and Claudine Chaouiya. A reduction of logical regulatory graphs preserving essential dynamical properties. In *Computational Methods in Systems Biology*, volume 5688 of *LNCS*, pages 266–280. Springer, 2009.

[10] Loïc Paulevé, Morgan Magnin, and Olivier Roux. Refining dynamics of gene regulatory networks in a stochastic π-calculus framework. In *Transactions on Computational Systems Biology XIII*, pages 171–191. Springer, 2011.

[11] Loïc. Paulevé, Morgan Magnin, and Olivier Roux. Static analysis of biological regulatory networks dynamics using abstract interpretation. *Mathematical Structures in Computer Science*, in press, 2012. Preprint: `http://loicpauleve.name/mscs.pdf`.

[12] Adrien Richard and Jean-Paul Comet. Necessary conditions for multistationarity in discrete dynamical systems. *Discrete Applied Mathematics*, 155(18):2403 – 2413, 2007.

[13] Adrien Richard, Jean-Paul Comet, and Gilles Bernot. *Modern Formal Methods and App.*, chapter Formal Methods for Modeling Biological Regulatory Networks, pages 83–122. 2006.

[14] Ozgur Sahin, Holger Frohlich, Christian Lobke, Ulrike Korf, Sara Burmester, Meher Majety, Jens Mattern, Ingo Schupp, Claudine Chaouiya, Denis Thieffry, Annemarie Poustka, Stefan Wiemann, Tim Beissbarth, and Dorit Arlt. Modeling ERBB receptor-regulated G1/S transition to find novel targets for de novo trastuzumab resistance. *BMC Systems Biology*, 3(1), 2009.

[15] Heike Siebert and Alexander Bockmayr. Incorporating time delays into the logical analysis of gene regulatory networks. In *Computational Methods in Systems Biology*, volume 4210 of *LNCS*, pages 169–183. Springer, 2006.

[16] René Thomas. Boolean formalization of genetic control circuits. *Journal of Theoretical Biology*, 42(3):563 – 585, 1973.

Parameterized verification of networks with many identical probabilistic processes

Paulin Fournier
ENS Cachan Antenne de Bretagne
paulin.fournier@inria.fr

Abstract

We introduce a model for networks of identical probabilistic timed processes, where the number of processes is a parameter. Each process is a probabilistic single-clock timed automaton and communicates with the others by broadcasting. On the one hand, we prove that most parameterized verification problems (*i.e.* verification independently of the number of processes) are undecidable in the static case, already in the untimed case. On the other hand, we show that presence of dynamism, via creation and extinction of processes, permits to recover decidability.

1 Introduction

Parameterized verification aims at verifying a system independently of its actual instantiation, that is in our case, independently of the number of processes involved. In [1], networks of many identical timed automata are introduced. For this model, safety properties are decidable if and only if the timed automata have a single clock. More recently, a series of papers, initiated with [5], investigates the parameterized verification of ad-hoc networks. The nodes in the network are modelled by finite automata (and later single-clock timed automata) that communicate through broadcast. The decidability status of reachability and coverability problems depends on the topology and its evolution.

Another aspect of verification are quantitative verification problems. A prominent class of quantitative systems is the one of probabilistic models, which are well known in the finite case. However, up to our knowledge, the parameterized verification (for the number of processes in the network) of probabilistic systems hasn't been investigated yet.

We introduce a modelling formalism that combines infinite-state space, due to an unknown number of processes in the network, and probabilistic behaviours. *Probabilistic timed networks* are formed of many identical probabilistic timed automata with a single clock, and interaction between the processes is modelled by message broadcasting. In a second step, in order to encompass mobility in the network, we also define *dynamic* probabilistic timed networks, where processes can disappear and be created.

On the one hand we prove that most parameterized problems are undecidable when the topology is static, already in the untimed case. On the other hand, in the dynamic case, we provide a transformation from probabilistic timed networks to a class of probabilistic lossy channel systems, and derive the decidability of the considered parameterized verification problems.

The rest of the paper is organized as follows. We define the model of probabilistic timed networks in Section 2 together with the parameterized verification problems we consider. Section 3 presents the undecidability in the static case. In Section 4 we adapt the model to the dynamic case and establish the decidability of the parameterized verification problems. We conclude by mentionning open questions and future work.

2 Modelling probabilistic protocols

Notations For E a finite or denumerable set, we write $\mathsf{Dist}(E)$ for the set of discrete probability distributions over E, that is, the set of functions $\delta : E \to [0,1]$ such that $\sum_{e\in E} \delta(e) = 1$. For E an arbitrary set, we write $\mathscr{M}(E)$ for the set of multisets over E, or equivalently the set of multiplicity functions $f : E \to \mathbb{N}$.

Given $\mathtt{x}$ a continuous clock the set of regions over $\mathtt{x}$, with a maximal constant $b \in \mathbb{N}_{>0}$, is denoted $\mathsf{Reg}(\mathtt{x})$. $\mathsf{Up} = \{\mathsf{id}, \mathsf{z}\}$ is the set of possible updates for clock $\mathtt{x}$: a clock update $\mathsf{up}(\mathtt{x})$ either resets the clock $\mathtt{x}$ to zero ($\mathsf{up} = \mathsf{z}$) or leaves it unchanged ($\mathsf{up} = \mathsf{id}$).

Model We define a new model for *probabilistic timed protocols* which are in fact probabilistic timed automata with a single clock, and with restrictions on the form of transition rules, to emphasize their communicative nature.

Definition 1. A *probabilistic timed protocol* is a tuple $\mathscr{P} = (Q, q_0, \mathtt{x}, \mathsf{I}, \Sigma, \Delta)$ where: Q is a finite set of control states, $q_0 \in Q$ is the initial state, $\mathtt{x}$ is a clock, $\mathsf{I} : Q \to 2^{\mathsf{Reg}(\mathtt{x})}$ is the invariant function, Σ is a finite message alphabet with a subset Σ_ε of internal actions, and Δ is the probabilistic discrete transition function, partitioned into: internal actions: $\Delta_\mathsf{i} : Q \times \mathsf{Reg}(\mathtt{x}) \times \Sigma_\varepsilon \to \mathsf{Dist}(Q \times \mathsf{Up})$, denoted $q \xrightarrow{g,\varepsilon,\mathsf{up},p} q'$ whenever $\delta(q', \mathsf{up}) = p$ with $\delta = \Delta_\mathsf{i}(q, g, \varepsilon)$; broadcasts: $\Delta_\mathsf{b} : Q \times \mathsf{Reg}(\mathtt{x}) \times !(\Sigma \setminus \Sigma_\varepsilon) \to Q \times \mathsf{Up}$, denoted $q \xrightarrow{g,!a,\mathsf{up}} q'$ whenever $(q', \mathsf{up}) = \Delta_\mathsf{b}(q, g, !a)$; receptions: $\Delta_\mathsf{r} : Q \times \mathsf{Reg}(\mathtt{x}) \times ?(\Sigma \setminus \Sigma_\varepsilon) \to Q \times \mathsf{Up}$, denoted $q \xrightarrow{g,?a,\mathsf{up}} q'$ whenever $(q', \mathsf{up}) = \Delta_\mathsf{r}(q, g, ?a)$.

A *probabilistic timed network*, written $\mathscr{P}^N$, is composed of $N \in \mathbb{N}_{>0}$ copies, called *processes*, of a probabilistic timed protocol $\mathscr{P}$.

The intuitive interpretation of a probabilistic timed network $\mathscr{P}^N$ is that N processes arranged in a clique execute the probabilistic timed protocol $\mathscr{P}$ simultaneously.

Remark 2. In our model, in each control state, several internal actions can be enabled, each giving rise to a probability distribution for the successor state, whereas broadcasts and receptions are deterministic. This is not a real restriction, since systems with nondeterministic and probabilistic choices for broadcast and receptions can be encoded in our model by introducing intermediary states and additional internal transitions.

Let us explain informally the semantics $[\![\mathscr{P}^N]\!]$ of a probabilistic timed network. A *configuration* of a probabilistic timed network $\mathscr{P}^N$ is a finite multiset $\gamma \in \mathscr{M}(Q \times \mathbb{R}_+)$ over the set of pairs composed of a control state and a real value for the clock. Each pair represents a process involved in the network (since we consider networks with N processes each configuration is composed of N pairs). We only consider *legal configurations* that satisfy the state invariants given in $\mathscr{P}$ *i.e.* $\gamma(q,x) > 0$ implies $x \in \mathsf{I}(q)$. The set of all such configurations is denoted Conf.

$[\![\mathscr{P}^N]\!]$, is given in terms of a timed Markov decision process over the configuration. The initial configuration γ_0, is defined as $\gamma_0(q_0, 0) = N$ and $\gamma_0(q,x) = 0$ otherwise. The transition relation is partitioned into time elapsing and discrete actions. Where time elapsing increments all the clocks values with a real value (and preserve the invariants) and discrete actions are defined informally as follows: from a configuration, first a process is selected nondeterministically, and second, a (broadcast or internal) transition enabled for that process is performed. Messages are broadcast to all other processes, whereas internal actions only affect the chosen process.

An *execution* in $[\![\mathscr{P}^N]\!]$ is a finite or infinite sequence $\gamma_0 \to \gamma_1 \to \gamma_2 \cdots$, where the transitions correspond to time elapsing, internal actions or communications.

A scheduler for a timed Markov decision process is a function σ that resolves the nondeterminism by choosing in each configuration an action: either time elapsing for all processes, or some element in the

support of the configuration together with an enabled internal action, or an enabled broadcast. In order to avoid some unrealistic time convergences, we also assume that, on any infinite execution, schedulers choose discrete actions infinitely often.

The probabilistic timed network $\mathscr{P}^N$ together with a fixed scheduler σ give rise to a Markov chain with state space Conf, and in which the probability measure over executions of $\mathscr{P}^N$, defined in a standard way, is written $\mathbb{P}_\sigma$.

Relevant verification questions We mainly focus on qualitative variants of reachability problems for networks of probabilistic protocols. For $q_f \in Q$, an execution ρ of $\mathscr{P}^N$ satisfies $\Diamond q_f$ (written $\rho \models \Diamond q_f$), if there exists a configuration γ along ρ that contains a process with control state q_f. We write $\mathbb{P}_\sigma(\mathscr{P}^N \models \Diamond q_f)$ for the probability under scheduler σ of the set of executions ρ with $\rho \models \Diamond q_f$, and further denote by $\mathbb{P}_{\min}(\mathscr{P}^N \models \Diamond q_f)$ (resp. $\mathbb{P}_{\max}(\mathscr{P}^N \models \Diamond q_f)$) for the minimum (resp. maximum) of these values among all possible schedulers.

We focus on the following relevant problems:

- $\text{REACH}^{>0}_{max}$: Does there exist N such that $\mathbb{P}_{max}(\mathscr{P}^N \models \Diamond q_f) > 0$? For q_f a bad state this question correspond to: is there a network size and an environment such that something bad can happen?

- $\text{REACH}^{=1}_{min}$: Does there exist N such that $\mathbb{P}_{min}(\mathscr{P}^N \models \Diamond q_f) = 1$? For q_f a good target state this problem ensure that for a well sized network, whatever what happen, q_f is almost surely reached.

- $\text{REACH}^{<1}_{min}$: Does there exist N such that $\mathbb{P}_{min}(\mathscr{P}^N \models \Diamond q_f) < 1$?[1] This question is relevant for negative answer: whatever the size of the network the state is almost surely reached.

3 Static number of processes

In the static case, where processes can not disappear or be created, the studied problems are undecidable except for the case $\text{REACH}^{>0}_{max}$. This case corresponds to reachability in the non probabilistic case, and is decidable [3]. The other problems are still undecidable even in the untimed restriction of the model.

This is proved by reduction of the halting problem (resp. the boundedness problem) of a IT 2-counter machine M (IT stands for infinitely testing: for an infinite run a non zero counter is tested to zero infinitely often, see [2]).

The idea of the reduction is the following: one process plays the role of the controller that keeps track of the control state in M, the other processes will encode the values of the counters. Increments and decrements are just changing the state of one of these processes. And zero tests are modelled by a probabilistic choice of the controller. In case of error in this choice we ensure that one process reaches an error state. Hence the only way to avoid error states is thus to faithfully simulate M.

With this construction, we can prove that $\text{REACH}^{<1}_{min}$ for the error state is equivalent to termination in M and that $\text{REACH}^{=1}_{min}$ is equivalent to counter boundedness in M, hence the undecidability.

4 Dynamic number of processes

In this section, we extend the model of probabilistic timed networks in order to take into account disparition and creation of processes. This dynamism is sensible for the application to wireless sensor networks where nodes can break down or run out of battery, but also be newly inserted or refill their battery.

Syntax and semantics A *dynamic probabilistic timed network* $(\mathscr{P}, N_0, \lambda_+, \lambda_-)$ is composed of probabilistic timed protocols $\mathscr{P}$ with initially N_0 copies, together with a creation rate $\lambda_+ \in (0,1)$ and a disparition rate $\lambda_- \in (0,1)$. After each discrete action and every time unit, each process disappears with probability λ_-, followed by the creation of k processes (in the initial control state with clock value 0) with probability $\lambda_+^k(1-\lambda_+)$.

[1] This is not the negation of the precedent problem because of the existantial quantifier.

Note that $\lambda_+ = \lambda_- = 0$ is forbidden, otherwise the number of processes is constant and we recover the model of probabilistic timed network from Definition 1.

The semantics of a dynamic probabilistic timed network is defined as for the static case, by a timed Markov decision process. The only difference is that it takes into account the distributions risen by the processes that disappear followed by the creation of new processes.

Mixed channel systems To show decidability we introduce mixed channel (MCS) system which are close to lossy channel system (LCS) and show that decidability in LCS still holds for MCS and that we can model probabilistic timed network in MCS.

Definition 3 (Mixed channel system)**.** A *mixed channel system* is a tuple $(S, \mathscr{C}, \Sigma, f, \Lambda)$ where S is a finite set of control states, $\mathscr{C}$ is a finite set of FIFO channels, $\Sigma = \Sigma_{\mathtt{r}} \cup \Sigma_{\mathtt{u}}$ is a finite set of messages partitioned into reliable and unreliable messages, $f \subseteq S \times 2^{\mathscr{C}} \times \big((\mathscr{C} \times \{!,?\} \times \Sigma) \cup \{\varepsilon\}\big) \times \mathsf{Dist}(S)$ is the probabilistic transition function, split into perfect transitions and gainy/lossy transitions: $f = f_{\mathtt{p}} \cup f_{\mathtt{1,g}}$, and Λ is the pair of creation and disparition rates.

Mixed channel systems are a mixture of channel systems with emptiness tests and unreliable channel systems with insertions and losses: first, not all messages can be lost, and second, not all transitions are unreliable. The model is undecidable in its full generality (it already encompasses perfect channel systems), however in the sequel we explain how to encode any dynamic probabilistic timed network into a mixed channel system with decidable reachability problem. This reduction will ensure the existence of a finite attractor and that predecessor operators are effectively computable and upward closed hence the decidability of reachability problems.

The semantics of a mixed channel system is an infinite-state Markov decision process where the states are configurations (s,w) formed of a control state $s \in S$ and a mapping $w : \mathscr{C} \to \Sigma^*$, describing the channel contents. The set of actions is composed of reads from and writes to the channels, and internal actions, potentially guarded by an emptiness test of a subset of the channels. The probability distributions are given by the probability of deletion and creation of lossy messages in the channels during gainy/lossy transitions.

In this Markov decision process, a scheduler $\mathscr{U}$ is a function that associates with each configuration (s,w) some enabled action (read, write or internal).

Reduction to mixed channel systems The principle of the reduction is to encode the configurations of $\mathscr{P}_\Lambda^{N_0}$ by channel contents. In order to keep the channel alphabet finite, we rather encode an abstraction of the configurations using the classical region abstraction for timed automata. In the region abstraction, the only relevant informations about a configuration are the state of each process, the integer part of their respective clock and the ordering of the fractional parts of their clocks. $\mathsf{Reg\_Abs}(\gamma)$ denotes this abstraction for γ. As an example $\mathsf{Reg\_Abs}((q,2.7),(q',1.3)) = (q,x) \wedge (q',x') \wedge 1 < x' < 2 < x < 3 \wedge \{x'\} < \{x\}$ (where $\{x\}$ denotes the fractional part of x).

The channel alphabet first contains unreliable messages encoding the control state together with the integer part of the clock (up to a maximal bound b): $\Sigma_1 = Q \times [0\,..\,b]$. The encoding of the ordering between fractional parts of clocks uses special reliable messages in $\Sigma_2 = \{0, D\}$ to delimit processes with equal clock fractional parts: any two messages in Σ_1 between two consecutive D's represent processes that share the same clock fractional part; also messages in Σ_1 between 0 and the first D represent processes with null fractional part; last, the channel contains encodings of fractional parts in increasing order. The channel contents thus belong to the regular language denoted by the expression $R = 0\Sigma_1^*(D\Sigma_1^+)^*$.

Let us give some examples for the encoding into channel contents of configurations $(q,x),(q',x')$ for various values of x, x'. The region $1 < x' < 2 < x < 3, \{x'\} < \{x\}$ is encoded by the channel content: $\mathbf{0D}(\mathbf{q}',\mathbf{1})\mathbf{D}(\mathbf{q},\mathbf{2})$, $0 < x < x' < 1$ by $\mathbf{0D}(\mathbf{q},\mathbf{0})(\mathbf{q}',\mathbf{0})$, $2 < x' < 3 = x$ by $\mathbf{0}(\mathbf{q},\mathbf{3})\mathbf{D}(\mathbf{q}',\mathbf{2})$, and $x = 2, x' = 1$ by $\mathbf{0}(\mathbf{q}',\mathbf{1})(\mathbf{q},\mathbf{2})$.

The reduction from dynamic probabilistic timed networks to mixed channel systems with probabilities is schematically represented in Figure 1. Given $\mathscr{P}_{\Lambda}^{N_0}$ a dynamic probabilistic timed network, the associated mixed channel system MCS is decomposed into three blocks and a recurrent state obs. The first block initializes the system, to encode the initial configuration: N_0 processes in $(q_0,0)$ so that the channel contents is $0(q_0,0)^{N_0}$. The system then contains two cycles around obs, one to encode time elapsing, and the other one to encode execution of an action. The action block is itself split into three consecutive sub-blocks: 1) the choice of a process and an enabled action, 2) the execution of the chosen action (modifying the configuration), and 3) a reordering of the channel contents to obtain an encoding of the configurations as explained above. The whole translation is done in such a way that the projection of an execution of the mixed channel system onto the configurations with control state obs corresponds to the region abstraction of an execution in the dynamic probabilistic timed network, and vice versa.

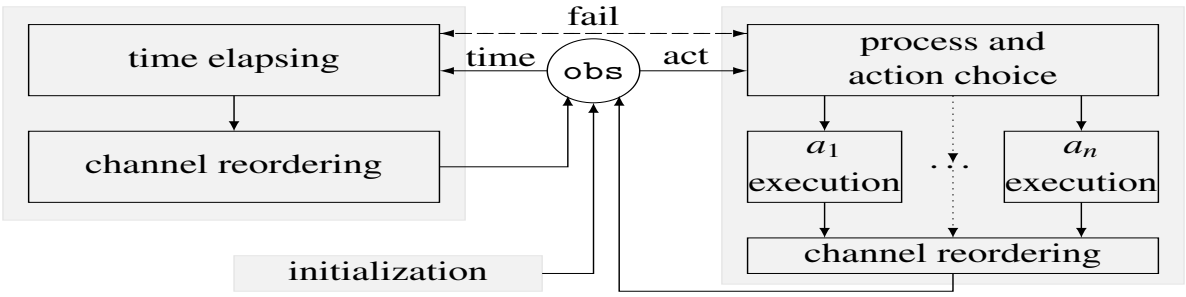


Figure 1: General figure of the reduction.

Let us now explain how each of the blocks works. From state obs two transitions are enabled: one leading to the time elapsing block, and the other to the process and action choice block, corresponding respectively to the decision for the scheduler to let time elapse or to perform an action.

Choice of a process and an action. The first step when the decision is taken to perform an action, is to non-deterministically choose a process and some enabled action. If no process has an enabled action, the channel system moves to the time elapsing block, via the dashed arrow labelled fail.

Execution of an action. Once an action is chosen, the corresponding execution block encodes the effect of performing this action. Depending on the nature of the action, internal action or broadcast, it affects the chosen process only or all processes. The update of the messages representing the processes in the channel takes part in two steps: first the control state of the processes are updated and marked to remember the associated clock update, and second the abstract clock values are updated and the messages reordered to take into account the clock updates. This second step happens in the next block.

In case of an internal action, the channel content is inspected to find the marked message corresponding to the chosen process (in the previous block). It is updated to its new control state and possibly a new abstract clock value according to the rule in the underlying probabilistic timed protocol (possibly with a probability distribution). Similarly, in case of a broadcast, the block modifies all messages (different from 0 and D) according to the transitions in $\mathscr{P}$.

Reordering of the channel contents. The goal of the *channel reordering* block is to transform the channel contents into one satisfying the regular expression R. Messages (for processes whose clock was just reset) are rewritten right after the marker 0 (with the new clock value 0) and the other messages are kept in place. In this block also, in order to keep the number of reliable messages as small as possible, multiple successive occurrences of D are replaced with a single D.

Time elapsing, or region change. Recall that we use the region abstraction to represent the timing

information of configurations. As a consequence, time elapsing is simulated by moving to the next time-successor region. We distinguish two cases. If some clocks have an integer value, then in the successor region, the ordering of fractional parts is unchanged, and no clocks have an integer value any-more. If all clocks have a non-null integer part, the time-successor is obtained when the clocks with maximal fractional part reach the next integer. This corresponds in the channel contents to rewriting the last D message as 0 and the 0 as D, while updating the new integer value for the concerned clocks (with the convention that $b+1=b$ for the maximal constant b). In both cases, time elapsing is only possible if the state invariants are not violated. This is checked before performing the region increment, and if time elapsing is not possible, the channel system moves to the action block via the fail transition.

Gainy/Lossy transitions. To represent the creations and disparitions of processes, in the channel system MCS, some of the transitions are gainy/lossy, that is, in $f_{1,g}$. Since disparitions and creations in $\mathscr{P}_{\Lambda}^{N_0}$ happen after each discrete action, and after each elapsed time unit, the transitions leaving the action execution blocks, and the time elapsing block (provided that a global clock, modelled in the channel and keeping track of global time, was reset) are declared to be gainy/lossy.

Properties and consequences There is a close connection between schedulers in the probabilistic network, and schedulers in the mixed channel systems, *i.e.* $\mathscr{P}_{\Lambda}^{N_0}$ and MCS behave the same, up to a region abstraction on one side, and a projection to configurations in obs on the other side.

With the properties that there exists a finite attractor, and second, the predecessor operators by action and time elapsing are effective and upward-closed for some well-quasi-ordering in MCS, we can show that probabilistic reachability problems are decidable. Hence $\text{REACH}_{max}^{>0}$, $\text{REACH}_{min}^{<1}$, and $\text{REACH}_{min}^{=1}$ are decidable for dynamic protocol networks.

5 Conclusion

We have studied qualitative parameterized verification problems for a model of networks of identical probabilistic timed processes. Interesting qualitative questions turn out to be undecidable in the static case, and become decidable under the assumption that processes can be created and disappear.

A relevant question is whether dynamic networks can be encoded into a decidable model with lower complexity than probabilistic lossy channel systems. An interesting research direction for future work, is to move from qualitative to quantitative questions. In particular, adapting existing techniques for Markov chains [2, 4] to Markov decision processes would allow one to approximate optimal reachability probabilities, to estimate maximum expected times to reachability, or to compute optimal values of the parameter.

References

[1] P. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theoretical Computer Science*, 290(1):241–263, 2003.

[2] P. A. Abdulla, N. Ben Henda, and R. Mayr. Decisive Markov chains. *Logical Methods in Computer Science*, 3(4), 2007.

[3] P. A. Abdulla, G. Delzanno, O. Rezine, A. Sangnier, and R. Traverso. On the verification of timed ad hoc networks. In *FORMATS*, volume 6919 of *LNCS*, pages 256–270. Springer, 2011.

[4] T. Brázdil and A. Kucera. Computing the expected accumulated reward and gain for a subclass of infinite markov chains. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 372–383. Springer, 2005.

[5] G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR'10*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010.

Implementation of Real-Time Systems: Theory and Practice

Aleksandra Jovanović
IRCCyN, Ecole Centrale de Nantes
Nantes, France
aleksandra.jovanovic@irccyn.ec-nantes.fr

Abstract

We study a parametric approach in the automata theory. We introduce parameters into timed game automata (TGA) framework and define a subclass for which the *reachability emptiness* problem is decidable. We then propose a semi-algorithm to symbolically compute the corresponding set of parameter valuations. We also study a parameter synthesis for parametric timed automata (PTA), and provide a subclass of PTA with a different restriction scheme: since in classical timed automata real-valued clocks are always compared to integers for all practical purposes, we also search for parameter values as bounded integers. In this way we obtain decidability for the most interesting problems. We introduce symbolic algorithms for the parameter synthesis that ensure the termination and have a practical advantage of giving the result as symbolic constraints between the parameters.

1 Introduction

Timed automata [1] and timed game automata [6] are one of the most often used frameworks for the analysis of real-time systems and control problems on real-time systems. A prerequisite for these methods is the availability of a complete model of the system. Thus, it can be difficult to use them at early design stages, when the whole system is not fully characterized.

It is sometimes possible to overcome this problem by using parameters for modeling values that are not fully characterized yet. To exploit such models, a parametric approach in automata theory must be used. The analysis of a parametric model produces symbolic constraints on the parameters that ensure the correctness of the system.

However, for general parametric formalisms such as parametric timed automata (PTA) [2], the existence of a parameter valuation, such that some state is reachable from the initial state, is undecidable and there is currently no algorithm that solves the synthesis problem of parameter values, except for severely restricted subclasses, whose practical usability is unclear.

It is then a challenging issue to define a subclass of parametric timed automata, which retains enough of its expressive power and such that, for both reachability and unavoidability properties, the existence of parameter values is decidable and for which there exist efficient symbolic synthesis algorithms.

We, as well, extend the model of timed game automata, such that it employs parameters. Obtained model is undecidable with respect to the reachability timed game, and we seek its subclass for which we can decide the existence of parameter valuation and, further, synthesize them.

2 Undecidability Results

Here we present some new undecidability results for the restricted classes of PTA that motivate the introduction of the new class of PTA presented in Section 3. Due to the lack of space, we omit the

proofs. We focus on two often studied problems:

- *EF-emptiness* problem: is the set of parameter valuations v, such that some location in the automaton is reachable, empty?
- *AF-emptiness* problem: is the set of parameter valuations v, such that all maximal runs of the automaton go through some location, empty?

Following the result from [2], where EF-emptiness is proven undecidable for PTA, we further restrict the problem by bounding parameter valuations ($k \leq v \leq K$ for $k, K \in \mathbb{Z}$) and prove (Theorem 1), that undecidability is retained.

Theorem 1. *Bounded EF-emptiness problem is undecidable for parametric automata.*

A L/U-automaton, that employs each parameter as either a lower or an upper bound on clocks, has been proposed in [4] as a subclass of PTA for which the EF-emptiness problem is decidable. However, by proving Theorem 2, we rule out the possibility of computing the solution set as a finite union of polyhedra.

Theorem 2. *If it can be computed, the solution to the EF-synthesis problem for L/U-automata cannot be represented using any formalism for which emptiness of the intersection with equality constraints is decidable.*

We now address a further subclass of L/U automata, U-automata, that allows only the upper bound parameters, and prove that AF-emptiness is undecidable (due to the fact that it is not based on the monotonicity property of L/U automata).

Theorem 3. *The AF-emptiness problem is undecidable for U-automata.*

These negative theoretical results show the need for the new class of PTA that will have the advantage of both decidability of the most interesting properties and the possibility of computation of the set of parameter valuations such that these properties are satisfied.

3 Integer Parameter Synthesis for Timed Automata

Currently the most useful known subclass of PTA, L/U automata have a syntactical restriction in the use of parameters. We therefore advocate for a different restriction scheme: since in classical timed automata, real-valued clocks are always compared to integers for all practical purposes, we also search for parameter values as bounded integers.

From a practical point of view, the subclass of PTA in such setting is not really restrictive since the temporal constraints for timed automata are usually defined on natural numbers. Nevertheless, this subclass is restrictive enough to make the problems we address decidable and to allow symbolic synthesis algorithms of parameter values. The case of integer parameters can be interesting only if we solve this problem symbolically - deriving the constraints on the parameters such that certain property is satisfied.

We can define several parameter-related problems on PTA that fall into two broad categories: *do there exist valuations for the parameters such that some property is satisfied* and *can we compute all of these valuations*? We have focused on reachability (EF-emptiness) and unavoidability (AF-emptiness) properties.

We first introduce symbolic semi-algorithms, EF and AF, to solve the synthesis problems in the general setting (possibly non integer valuations) that are based on a straightforward extension of the symbolic zone-based state-space exploration that is ubiquitous for timed automata. For $S = (l, Z)$, when non-ambiguous, we use S in place of l or Z to simplify the writing.

For EF we aggregate the valuations found when reaching the locations in G:

$$\mathsf{EF}_G(S,M) = \begin{cases} S_{|P} & \text{if } S \in G \\ \emptyset & \text{if } S \in M \\ \bigcup_{\substack{e \in E \\ S' = \mathsf{Succ}(S,e)}} \mathsf{EF}_G\big(S', M \cup \{S\}\big) & \text{otherwise.} \end{cases}$$

$$\mathsf{AF}_G(S,M) = \begin{cases} S_{|P} & \text{if } S \in G \\ \emptyset & \text{if } S \in M \\ \bigcup_{\substack{e \in E \\ S' = \mathsf{Succ}(S,e)}} \mathsf{AF}_G\big(S', M \cup \{S\}\big) \cap \bigcap_{\substack{e' \in E \\ e' \neq e \\ S'' = \mathsf{Succ}(S,e')}} \mathsf{AF}_G\big(S'', M \cup \{S\}\big) \cup (\mathbb{R}^P \setminus S''_{|P}) & \text{otherwise.} \end{cases}$$

In both algorithms, conditions are evaluated from top to bottom and M represents a *passed list* of symbolic states. It records the symbolic states that have already been explored on a given path. Initially, M is empty and, for the EF-synthesis problem, for instance, the invocation of EF is, for the PTA $\mathscr{A}$ and a subset of its locations G: $\mathsf{EF}_G(\mathsf{Init}(\mathscr{A}), \emptyset)$.

3.1 Extension of the semi-algorithms for integer synthesis problem

In order to compute the integer valuations, the semi-algorithms have to be modified. For that we introduce the notion of integer hull.

Let $n \in \mathbb{N}$ and let Y be a subset of $\mathbb{R}^n$. We denote by $\mathsf{Conv}(Y)$ the *convex hull* of Y, i.e. the smallest convex set containing Y. $\mathsf{IntVects}(Y)$ denotes the subset of all elements of Y with integer coordinates. We call those elements *integer valuations*. The *integer hull* of Y, denoted by $\mathsf{IntHull}(Y)$ is the smallest convex set containing all the integer vectors of Y, i.e. $\mathsf{IntHull}(Y) = \mathsf{Conv}(\mathsf{IntVects}(Y))$.

We extend IntVects to symbolic states (l,Z) (consisting of a location l and a set of valuations on clocks and parameters Z) by: $\mathsf{IntVects}((l,Z)) = (l, \mathsf{IntVects}(Z))$ and extend likewise all the other operators on valuation sets.

We have shown (here we omit numerios lemmas, due to the lack of space) that it is sufficient to consider the integer hulls of the (valuations in the) symbolic states.

We therefore consider the semi-algorithm IEF (resp. IAF) obtained from EF (resp. AF) by replacing all occurrences of the symbolic state successor operator Succ by ISucc with $\mathsf{ISucc}((l,Z),e) = \mathsf{IntHull}(\mathsf{Succ}(l,Z),e)$. We also extend ISucc to edge sequences in the same way as for Succ. Now, we can state the main result of this subsection: IEF and IAF are correct semi-algorithms for their respective *integer* synthesis problems.

Theorem 4. *For any PTA $\mathscr{A}$ and any subset of its locations G, upon termination, $\mathsf{IEF}_G(\mathsf{Init}(\mathscr{A}), \emptyset)$ (resp. $\mathsf{IAF}_G(\mathsf{Init}(\mathscr{A}), \emptyset)$) is the solution to the integer EF-synthesis (resp. AF-synthesis) problem for PTA $\mathscr{A}$ and set of locations to reach G.*

In order to ensure the termination of the algorithm, we consider that we are searching the bounded integer parameter valuations. Given some $N \in \mathbb{Z}$, we search for valuations in $[-N,N]$. The algorithm accepts the bound on the parameters in the initial symbolic state, and therefore in the whole computation, which ensures the termination of the algorithm.

Theorem 5. *For any $M,N \in \mathbb{N}$, any PTA $\mathscr{A}$ and any subset of its locations G, Algorithms $\mathsf{IEF}_G(\mathsf{Init}_{M,N}(\mathscr{A}), \emptyset)$ and $\mathsf{IAF}_G(\mathsf{Init}_{M,N}(\mathscr{A}), \emptyset)$ terminate.*

Considering bounded integers as values for the parameters seems like a big theoretical restriction but it offers the algorithm for the synthesis problems for the expressive model of PTA, which is of great practical interest. In practice, this is not such a big restriction for the modeler because the integer constants are the natural choice. Also, the symbolic approach allows for choosing very big bounds on parameters.

4 Parametric Timed Game Automata

Timed game automata are used for modeling and analyzing the control problems on real-time systems. A timed game automaton is essentially a timed automaton whose set of actions is partitioned into controllable and uncontrollable actions. Two players, a controller and an environment, choose at every instant one of the available actions from their own sets and the game progresses. Finding a winning strategy for the controller consists in determining when and which of the controllable actions should be taken such that, regardless of what the environment does, the system ends up in a desired location. We introduce parameters into timed games and obtain a new model called parametric timed game automaton [5].

For parametric timed automata (PTA), [2], the EF-emptiness problems asks whether the set of parameter valuations, such that a certain state is reachable from the initial state, is empty. We define the EF-emptiness problem for parametric timed games (PGA).

Definition 6. EF-emptiness problem for PGA is the problem of determining whether the set of parameter valuation, such that there is a strategy for the controller to reach the desired state, is empty.

The EF-emptiness problem for PTA is known to be undecidable, [2]. As PGA extend PTA, this problem for PGA is undecidable as well. It is therefore interesting to find a class of PGA for which the EF-emptiness problem is decidable.

4.1 L/U Game Automata

We introduce the new class of parametric timed games, called L/U game automata, [5]. The parameters are partitioned into two sets. The first set P^l contains parameters that are used as lower bounds in the guards on the controllable transitions and as upper bounds in the guards of the uncontrollable transitions. The parameters from the other set, P^u, are used as upper bounds in the controllable and lower bounds in the uncontrollable transitions. This is a natural way of making the controller more powerful by restricting possible behaviors of the environment (and vice-versa). We assume that invariants are non-parametric constraints.

For this class of timed games, we prove that the existence of parameter valuation, such that there is a strategy to reach the desired state, is decidable.

Theorem 7. *The EF-emptiness problem for L/U game automata is decidable.*

In terms of control it may not be very realistic to be allowed to forbid uncontrollable transitions using the values of their parameters. We will now explore the case in which we nonetheless impose that the parameter valuations never set the guards of the uncontrollable transitions to false: we can restrict their behavior but not to the point of uniformly forbidding the transition.

Consequently, all the guards on the uncontrollable transitions that contain a parameter as a lower (resp. upper) bound have to contain a constant as a non-strict upper (resp. lower) bound. Non-strict inequalities are mandatory so that a clock can take the value equal to the constant as a single time point in the emptiness test. The guards on the controllable transitions have no other restriction than the L/U

condition. We also limit the parametric linear expression in the constraints of uncontrollable transitions to just one parameter.

Definition 8 (Restricted L/U game automata)**.** A restricted L/U game automaton is a L/U game automaton in which the guards of the uncontrollable transitions are constraints of the form $k \leq x \leq Ka$ or $Ka \leq x \leq k$, where x is a clock, a a parameter, k a rational number and K a natural number.

We state the decidability of the EF-problem for restricted L/U game automata in Theorem 9.

Theorem 9. *The EF-emptiness problem for restricted L/U game automata is decidable.*

4.2 Algorithm for Parameter Synthesis

In [3], the authors present a symbolic on-the-fly algorithm for solving timed reachability games. This algorithm consists of a forward computation of the simulation graph and a back-propagation of information of winning states. Upon the termination, the set of winning states of the simulation graph, from which we can extract the winning strategy, is obtained.

We have extended this algorithm for parameter synthesis, in order to compute the set of parameter valuations, together with the set of winning states.

To modify this algorithm so as to compute parameter valuations, we have used an extended notion of symbolic state in which we have a location and a *parametric zone* - a polyhedron constraining both clocks and parameters together, i.e., a set of pairs (w, v) satisfying a parametric clock constraint, where w is a clock valuation and v a parameter valuation.

The algorithm requires some specific operations on symbolic states. We have straightforwardly extended them for this extended notion of symbolic state.

The termination of our parametrization of the algorithm is not guaranteed. In the case of termination however, if the initial state belongs to a set of winning states, the correct set of constraints on the parameters is obtained and a strategy can be extracted from the set of winning states.

5 Conclusions and Future Work

There are several things planned for the future work. First, there is a question of time-bounded reachability problem for PTA. We want to prove its decidability for certain classes of PTA and write the algorithm for the parameter synthesis. Further, we want to extend hybrid automata with uncontrollable transitions and define necessary operators for solving timed game on this new model. For the parametric timed games, it could be interesting to look for the less restrictive subclasses and implement the proposed parametrized algorithm. Also, the algorithm for the integer parameter synthesis could be extended for the parametric timed games.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[3] F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, 2005.

[4] Thomas Hune, Judi Romijn, Marielle Stoelinga, and Frits Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002.

[5] Aleksandra Jovanović, Sébastien Faucou, Didier Lime, and Olivier H. Roux. Real-Time Control with Parametric Timed Reachability Games. In *11th International Workshop on Discrete Event Systems (WODES'12)*, Guadalajara, Mexico, October 2012. IFAC.

[6] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS '95*, 1995.

Model Checking Concurrent Systems with Unboundedly Many Processes Using Data Logics

Ahmet Kara
TU Dortmund, Germany
ahmet.kara@cs.tu-dortmund.de

Abstract

We investigate the model checking problem for Dynamic Communicating Automata (DCA) [5] using the data logics Freeze LTL ($\text{LTL}^{\downarrow}$) [9] and Basic Data LTL ($\text{BD}-\text{LTL}$) [16]. DCA model concurrent systems with unboundedly many processes which can communicate with each other. Data Logics express properties of data words, i.e. words where every position carries some symbols from a finite set and some data values from an infinite set. We interprete the process IDs in the runs of DCA as data values and consider the following model checking problem: Given a DCA and a formula of $\text{LTL}^{\downarrow}$ or $\text{BD}-\text{LTL}$, is the formula satisfied on all runs of the DCA? While this problem is undecidable in general we find some restrictions of DCA such that the considered problem gets decidable.

1 Introduction

Logics on data words, i.e. words where every position is labelled by some symbols from a finite set and some data values from an infinite set were investigated intensively in recent years [9][2][8][16][13][12]. In this work we concentrate on the data logics Freeze LTL ($\text{LTL}^{\downarrow}$) [9] and Basic Data LTL ($\text{BD}-\text{LTL}$) [16] which are incomparable in expressive power.

The logic $\text{LTL}^{\downarrow}$ is an extension of classical LTL by register variables. It allows to store a data value of a current word position in some register x (by $\downarrow x$) and to test at some other position whether some data value of that position and the one stored in x are the same (by $\uparrow x$). For example, the formula $\text{G}(a \rightarrow \downarrow^1 x.\text{F}(b \wedge \uparrow^2 x))$ expresses that every symbol a is followed by some b such that the first data value of the a-position and the second data value of the b-position are equal.

The logic $\text{BD}-\text{LTL}$ is also an extension of LTL. Besides the usual navigation on consecutive positions it allows the navigation along positions carrying the same data value. For instance, the formula $\text{FC}_1(\text{G}\neg b)$ expresses that there is a position p such that on all following positions carrying the same data value as the first data value of p there does not occur any b.

In most of the papers dealing with data logics, besides the area of XML, system verification is given as an important motivation. Data words can be used to represent runs of systems where an unbounded number of processes can occur and interact. In this context the data values of a word position represent the IDs of the processes performing an action and the symbols from the finite set represent the performed actions. Then, formulas of data logics can be used to express properties of system runs.

Our main motivation is to consider the model checking problem with data logics on models which

(1) describe the behavior of concurrent systems with unboundedly many processes, and

(2) produce system runs which can be represented by data words when the process IDs occurring in the runs are interpreted as data values.

Even though system verification is often given as an important motivation for data logics, as far as we know, until now the most investigated question in the area of data logics is rather satisfiability than model checking. Nevertheless, we would like to give some notable exceptions. In [11] and [10] the authors interpret runs of one-counter automata as data words and investigate the model checking problem on one-counter automata with data logics. While counter automata have many applications in, e.g. timed systems [17] and programs with pointer variables [6], the authors of [11] and [10] do not motivate counter values primarily as a representation for process IDs in systems with many processes. An other remarkable paper which tackles exactly the same question which motivates our work is [3]. There, the authors consider the model checking problem for data multi-pushdown automata (DMPA) with data logics. DMPA generate data words and they are indeed suitable to model concurrent programs with dynamic process creation.

In this work we investigate the model checking problem for dynamic communicating automata (DCA) [5] with $\text{LTL}^{\downarrow}$ and $\text{BD}-\text{LTL}$. DCA seem to be a very simple and convenient model to describe systems with unboundedly many processes. They are an extension of communicating finite state machines [7] and allow the creation of fresh processes and the communication between them through communication channels. A process maintains the communication to another process by storing its process ID in one of its process variables.

Work in progress and first insights It follows from [7] that the model checking problem on DCA with unbounded message channels with these logics is in general not decidable. This even holds for DCA with a finite number of processes. We can show that model checking of DCA remains undecidable if we use bounded channels but two process variables for every process.

One of our positive results is that model checking of DCA with $\text{BD}-\text{LTL}$ is decidable when communication channels are bound and every process uses only one process variable.

On the other hand, model checking with $\text{LTL}^{\downarrow}$, even on automata where communication among processes is not allowed (and which we therefore just call dynamic automata), is not decidable. However, we can show that on faulty dynamic automata, i.e. dynamic automata where unexpected processes can occur, model checking with $\text{LTL}^{\downarrow}$ gets decidable.

These are just our first insights in this area and further directions are mentioned in the last section.

2 Data Words

Let *Prop* be a *finite* set of propositions and $\mathscr{D}$ an *infinite* set of data values. For $m \geq 1$ an *m-dimensional data word* over *Prop* and $\mathscr{D}$ is a finite sequence $(P_1, \vec{d}_1), \ldots, (P_n, \vec{d}_n)$ of tuples such that for every $i \in \{1, \ldots, n\}$ the set P_i is a subset of *Prop* and $\vec{d}_i$ is a nonempty vector of elements from $\mathscr{D}$ of length at most m. In this work we are concerned with 1-dimensional and 2-dimensional data words.

3 Data Logics

Freeze LTL

Freeze LTL ($\text{LTL}^{\downarrow}$) was introduced in [9] and allows to store data values in registers and to test whether the data value in some register is equal to a data value at some position. Here, we only deal with $\text{LTL}^{\downarrow}$ with one register on 1-dimensional data words. Formulas of this logic are constructed according to the following grammar [1]:

$$\varphi := p \mid \uparrow \mid \neg\varphi \mid \varphi \vee \varphi \mid \downarrow.\varphi \mid \text{X}\varphi \mid \varphi_1 \text{U} \varphi_2$$

[1] As only one register is used we omit the register variable in the syntax.

whith $p \in Prop$.

Formulas of LTL$^{\downarrow}$ are evaluated with respect to a 1-dimensional data word $w = (P_1, d_1), \ldots, (P_n, d_n)$, a position $i \in \{1, \ldots, n\}$ on w and a partial register mapping $\nu : \{1\} \to \mathscr{D}$. In the definition of the semantics we omit the boolean cases:

- $(w, i, \nu) \models p$ if $p \in P_i$
- $(w, i, \nu) \models \uparrow$ if $\nu(1)$ is defined and $\nu(1) = d_i$
- $(w, i, \nu) \models \downarrow.\psi$ if $(w, i, \{1 \mapsto d_i\}) \models \psi$
- $(w, i, \nu) \models \mathrm{X}\psi$ if $i < n$ and $(w, i+1, \nu) \models \psi$
- $(w, i, \nu) \models \psi_1 \mathrm{U} \psi_2$ if there exists $j \geq i$ such that $(w, j, \nu) \models \psi_2$ and $(w, k, \nu) \models \psi_1$ for all $i \leq k < j$

A 1-dimensional data word w satisfies a LTL$^{\downarrow}$ formula φ if $(w, 1, \nu) \models \varphi$ where ν is the register mapping where $\nu(1)$ is undefined.

In [9] it is shown that the satisfiability problem for LTL$^{\downarrow}$ is decidable with non-primitive recursive complexity. It is also proven that adding past time operators or a second register causes undecidability.

Basic Data LTL

Basic Data LTL (BD – LTL) was introduced in [16] and allows the navigation along positions carrying the same data value. In this work we only consider BD – LTL over 2-dimensional data words. Among the BD – LTL subformulas we distinguish between position formulas φ and class formulas ψ whose syntax is defined as follows[2]:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathrm{X}\varphi \mid \mathrm{X}_{-}\varphi \mid \varphi \mathrm{U} \varphi \mid \varphi \mathrm{U}_{-}\varphi \mid \mathrm{C}_j \psi$$

$$\psi ::= \varphi | @_j | \neg\psi \mid \psi \vee \psi \mid \mathrm{X}^{=}\psi \mid \mathrm{X}^{=}_{-}\psi \mid \psi \mathrm{U}^{=}\psi \mid \psi \mathrm{U}_{-}{}^{=}\psi$$

with $p \in Prop$ and $j \in \{1, 2\}$.

The logic BD – LTL consists of all position formulas.

Position formulas are evaluated with respect to a 2-dimensional data word and a word position and class formulas are evaluated additionally with respect to a data value. Let $w = (P_1, \vec{d}_1), \ldots, (P_n, \vec{d}_n)$ be a 2-dimensional data word, i a position in w and $d \in \mathscr{D}$ a data value. In the semantics definition we again omit the boolean cases and the past operators X_{-} and U_{-} which are duals of X and U.

- $(w, i) \models p$ if $p \in P_i$
- $(w, i) \models \mathrm{X}\chi$ if $i < n$ and $(w, i+1) \models \chi$
- $(w, i) \models \chi_1 \mathrm{U} \chi_2$ if there exists a $k \geq i$ with $(w, k) \models \chi_2$ and $(w, \ell) \models \chi_1$ for all $i < \ell \leq k$
- $(w, i) \models \mathrm{C}_j \psi$ if $\vec{d}_i(j)$ is defined and $(w, i, \vec{d}_i(j)) \models \psi$
- $(w, i, d) \models \varphi$ if $w, i \models \varphi$ (for a position formula φ)
- $(w, i, d) \models @_j$ if $\vec{d}_i(j) = d$
- $(w, i, d) \models \mathrm{X}^{=}\psi$ if there exists a $k > i$ and $j \in \{1, 2\}$ such that $\vec{d}_k(j) = d$, $(w, k, d) \models \psi$ and for all ℓ with $i < \ell < k$ there is no $j \in \{1, 2\}$ with $\vec{d}_\ell(j) = d$

[2] Compared to [16] we give here a restricted fragment of BD – LTL where the explicit navigation to some position with a different data value is not included.

- $(w,i,d) \models \psi_1 \mathrm{U}^{=} \psi_2$ if there exists a $k \geq i$ and a $j \in \{1,2\}$ with $\vec{d_k}(j) = d$ and $(w,k,d) \models \psi_2$ and for all ℓ with $i \leq \ell < k$ such that here exists a $j \in \{1,2\}$ with $\vec{d_\ell}(j) = d$ it holds $(w,\ell,d) \models \psi_1$

A BD – LTL formula φ is satisfied by a data word w if $(w,1) \models \varphi$.

In [16] it is shown that the satisfiability problem for BD – LTL is decidable. The authors prove that the problem is as hard as the nonemptiness problem for multi-counter automata [15]. The precise complexity of the latter is not known.

4 Dynamic Communicating Automata

As an extension of [5], in the definition of *dynamic communicating automata (DCA)* given here we assign propositions to automata states. A DCA $\mathscr{A}$ over *Prop* is a tuple $(X, M, Q, q_0, s, \delta, F)$ where

- $X = \{x_1, \ldots, x_n\}$ is a finite set of process variables,
- M is a finite set of messages,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $s : Q \to 2^{Prop}$ maps each state to a finite set of propositions
- $\delta \subseteq Q \times Act \times Q$ is a finite set of transitions and
- $F \subseteq Q$ is the set of final states.

The set *Act* is a finite set of actions of $\mathscr{A}$ including the actions $\{spawn(x,x',q) \mid x,x' \in X, q \in Q\} \cup \{send(x,m,\vec{x}), receive(x,m,\vec{x}) \mid x \in X, m \in M$ and $\vec{x}$ is a vector of process variables of length $n\}$.

We describe the semantics of DCA informally. The IDs of the processes contained in the system described by a DCA $\mathscr{A}$ are chosen from $\mathscr{D}$. A run of $\mathscr{A}$ is represented by a data word over *Prop* and $\mathscr{D}$. Every process holds his own process variables $x_1, \ldots, x_n$. A process can only perform sends and receives to processes whose IDs are stored in its registers. At the beginning, the system described by $\mathscr{A}$ contains only one process. If a process d performs $spawn(x,x',q)$ leading to state $\hat{q}$, a new process with a fresh ID d' starting at state q is created, d' is stored into variable x of process d and d is stored into variable x' of process d'. Within the run this action is represented by a position labelled by $(s(\hat{q}) \cup \{spawn\}, (d,d'))$. Assume that the ID d' is stored in a process variable x of process d and d executes $send(x,m,\vec{x})$ for some vector $\vec{x}$ such that the action leads to some state $\hat{q}$. Then, the message $m(d_1, \ldots, d_n)$ where $d_1, \ldots, d_n$ are the contents of the variables in $\vec{x}$ in process d, is put into the fifo-channel between d and d'. The action is represented by $(s(\hat{q}) \cup \{send\}, (d,d'))$. The vector $\vec{x}$ in a receive-action decides how the IDs taken from the channel are distributed to the variables of the receiving process. The representation of receive-actions in runs is analogue to that of send-actions. Every other action a of a process d leading to a state $\hat{q}$ is represented by $(s(\hat{q}) \cup \{a\}, d)$. A run of $\mathscr{A}$ is accepting if all processes end up in accepting states.

DCA with n process variables are also called *n-variable-DCA*. If a DCA does not contain any send- or receive-actions it is just called *dynamic automaton* (DA). Obviously, DA generate 1-dimensional data words as runs. DA where in each execution step new processes in arbitrary states can arise unexpectedly are called *faulty DA*.

5 Model Checking Dynamic Communicating Automata

The model checking problem for dynamic communicating automata and a data logic $\mathscr{L}$ is defined as follows:
Given: A DCA $\mathscr{A}$ and a formula φ of $\mathscr{L}$

Question: Is φ satisfied on all accepting runs of $\mathscr{A}$?
It easily follows from [7] that the model checking problem on DCAs with unbounded channels is not decidable even with LTL where no access to data values is possible. To show this, a finite number of processes suffices.

Remark 1. The model checking problem for DCA and LTL is not decidable.

5.1 Model Checking with BD – LTL

For DCA without channel capacities and two process variables per process the model checking problem with BD – LTL remains undecidable.

Theorem 2. *The model checking problem for 2-variable-DCA without channel capacities and* BD – LTL *is undecidable.*

Proof idea. By reduction from the nonemptiness problem for 2-counter automata [18]. The reduction makes use of the fact that with 2-variable-DCA it is possible to construct an unbounded chain of processes where every two neighbored processes are linked to each other by their process variables, and to send messages forth and back among these processes. [3] □

With 1-variable-DCA it is not possible to relate more than two processes with each other by their process variables at the same time. Thus, the crucial property mentioned in the above proof idea is not given for 1-variable-DCA. Indeed, for 1-variable-DCA model checking with BD – LTL is decidable.

Theorem 3. *The model checking problem for 1-variable-*DCA *with bounded channels and* BD – LTL *is decidable.*

Proof idea. By reduction to the non-emptiness problem for multi-counter-automata [15]. □

5.2 Model Checking with Freeze LTL

For $\text{LTL}^{\downarrow}$ the situation is much more worse. Even on DA (no send- and receive-actions) model checking is undecidable with $\text{LTL}^{\downarrow}$.

Theorem 4. *The model checking problem for DA and* $\text{LTL}^{\downarrow}$ *is undecidable.*

Proof idea. By reduction from the non-emptiness problem for 2-counter automata. □

However, model checking with $\text{LTL}^{\downarrow}$ gets decidable if we consider faulty DA.

Theorem 5. *The model checking problem for faulty DA and* $\text{LTL}^{\downarrow}$ *is decidable.*

Proof idea. By a reduction to the reachability problem in well-structured transition systems [14]. □

6 Further Directions

We indicated in Chapter 5 that the undecidability proof of Theorem 2 relies on the fact that with 2-variable-DCA messages can be sent forth and back among an unbounded number of processes. We can show that for DCAs (no matter how many process variables are used) where at each time only a bounded number of processes can stay in a communication relation, model checking with BD – LTL is decidable. It would be interesting to find out how DCA can be restricted such that this property holds on all possible runs.

It is known that many problems which are undecidable on systems with a finite number of processes but unbounded communication channels get decidable when *lossy* channels, i.e. channels where messages can get lost, are considered [1]. We would like to find out whether the model checking problem on general DCA is decidable if lossy channels are used.

[3] A detailed proof can be found in [4].

References

[1] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

[2] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.

[3] Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In Lars Birkedal, editor, *FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012.

[4] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara, and Thomas Schwentick. Dynamic Communicating Automata and Branching High-Level MSCs. Research Report LSV-12-20, LSV, November 2012.

[5] Benedikt Bollig and Loïc Hélouët. Realizability of dynamic MSC languages. In Farid M. Ablayev and Ernst W. Mayr, editors, *CSR*, volume 6072 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2010.

[6] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531. Springer, 2006.

[7] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[8] Stéphane Demri, Deepak D'Souza, and Régis Gascon. A decidable temporal logic of repeating values. In Sergei N. Artëmov and Anil Nerode, editors, *LFCS*, volume 4514 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2007.

[9] Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. In *LICS*, pages 17–26, 2006.

[10] Stéphane Demri, Ranko Lazic, and Arnaud Sangnier. Model checking freeze LTL over one-counter automata. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 490–504. Springer, 2008.

[11] Stéphane Demri, Ranko Lazic, and Arnaud Sangnier. Model checking memoryful linear-time logics over one-counter automata. *Theor. Comput. Sci.*, 411(22-24):2298–2316, 2010.

[12] Diego Figueira. A decidable two-way logic on data words. In *LICS*, pages 365–374. IEEE Computer Society, 2011.

[13] Diego Figueira and Luc Segoufin. Future-looking logics on data words and trees. In Rastislav Královic and Damian Niwinski, editors, *MFCS*, volume 5734 of *Lecture Notes in Computer Science*, pages 331–343. Springer, 2009.

[14] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[15] Jay L. Gischer. Shuffle languages, petri nets, and context-sensitive grammars. *Commun. ACM*, 24(9):597–605, 1981.

[16] Ahmet Kara, Thomas Schwentick, and Thomas Zeume. Temporal logics on words with multiple data values. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 8 of *LIPIcs*, pages 481–492. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[17] Ugo Dal Lago, Angelo Montanari, and Gabriele Puppis. On the equivalence of automaton-based representations of time granularities. In *TIME*, pages 82–93. IEEE Computer Society, 2007.

[18] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

Compositional reasoning about concurrent libraries on the axiomatic TSO memory model

Artem Khyzha
IMDEA Software Institute
artem.khyzha@imdea.org

Alexey Gotsman
IMDEA Software Institute
alexey.gotsman@imdea.org

Abstract

Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms, which has recently started to become adopted for weaker consistency guarantees provided by hardware and software platform. In this paper, we present the first definition of linearizability on the axiomatically formulated Total Store Order weak memory model, implemented by x86 processors. We establish that our definition is a correct one in the following sense: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. This allows abstracting from the details of the library implementation while reasoning about the client.

1 Introduction

Modern multiprocessor architectures, such as Intel x86 [5], IBM POWER [6, 4] and ARM, provide memory consistency models that are *weaker* than the classical sequential consistency (SC). What makes these models different is that they do certain relaxations to the order of memory accesses, which make program execution not sequentially consistent. Relying on relaxations enables implementing programs more efficiently, but leads to counter-intuitive behaviours in many cases, so programming on weak memory models can be subtle and error-prone.

Compositional reasoning about programs on the weak memory models requires a new formalisation for **correctness** of program components. Correctness of concurrent libraries is commonly formalised by the notion of *linearizability* [3], which fixes a certain correspondence between the library and its (usually sequential) abstract specification with methods implemented atomically. Unfortunately, the classical definition of linearizability is only appropriate for sequentially consistent (SC) memory models, in which accesses to shared memory occur in a global-time linear order.

In this paper we suggest an approach for compositional reasoning on a weak memory model of Total Store Order (TSO), implemented by x86 processors [5] (Sections 2, 3). TSO allows the store buffer optimisation implemented by modern multiprocessors: writes performed by a processor are buffered in a processor-local store buffer and are flushed into the memory at some later time.

A consequence of the store buffer optimisation is that on TSO, given two memory locations x and y initially holding 0, if two CPUs respectively write 1 to x and y and then read from y and x, as in the following program, it is possible for both to read 0 in the same execution:

$$\begin{array}{c} \{x = y = 0\} \\ x = 1; \quad \| \quad y = 1; \\ b = y; \quad \| \quad a = x; \\ \{a = b = 0\} \end{array}$$

This happens when the reads from y and x occur before the writes to them have propagated from the store buffers of the corresponding CPUs to the memory. To exclude such behaviours, TSO processors provide special instructions, called *memory fences*, that force the store buffer of the corresponding CPU to be flushed completely before executing the next instruction. Adding memory fences after the writes to x and y in the above program would make it produce only SC behaviours.

In this paper, we present the definition of linearizability on a weak memory model of TSO, which is different from a classic definition due to the store buffer relaxation. Usually the semantics of weak memory models is described in operational or axiomatic setting, and in this work we choose the axiomatic way. While operational model is more intuitive, the axiomatic semantics is more abstracted from a particular implementation and in some situations is easier to reason about.

We show that our definition of linearizability is a right one in the sense that it validates what we call the Abstraction Theorem (Theorem 4, Section 4): while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. Abstraction theorem has a practical value as a compositional reasoning and verification technique: it enables abstracting from the details of the library implementation while reasoning about its client, despite subtle interactions between the two caused by the weak memory model.

2 Preliminaries

The most intuitive way to explain TSO is to define its operational semantics using an abstract machine. In the following, we informally present the operational model, previously described in [5], and then in Section 3 we formally define axiomatic semantics. Due to space constraints we do not provide a proof of their equivalence.

Programming language. We assume that the memory consists of locations $\mathsf{Loc} = \{1, 2, \ldots\}$ containing values $\mathsf{Val} = \mathbb{Z}$. We consider programs in the following core language:

$$C ::= \alpha \mid C; C \mid C + C \mid C^* \mid m \qquad L ::= \{m = C_m \mid m \in M\} \qquad C(L) ::= \mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$$

A program consists of a ***library*** L implementing methods $m \in \mathsf{Method}$ and its ***client*** $C_1 \parallel \ldots \parallel C_n$, given by a parallel composition of threads (for simplicity, in this paper we suppose that all threads are bijectively mapped to a set of CPUs). Threads are indexed by $\mathsf{ThreadID} = \{1, \ldots, n\}$. The commands include primitive commands $\alpha \in \mathsf{PComm}$, method calls $m \in \mathsf{Method}$, sequential composition $C; C'$, non-deterministic choice $C + C'$ and iteration C^*. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands, the latter can be defined in the language as syntactic sugar.

We assume that every method accepts a single parameter and returns a single value. Parameters and return values are passed by every thread via distinguished locations in memory, denoted $\mathsf{param}_t, \mathsf{retval}_t \in \mathsf{Loc}$ for $t \in \mathsf{ThreadID}$. The rest of memory locations are partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc} \uplus \{\mathsf{param}_t, \mathsf{retval}_t \mid t \in \mathsf{ThreadID}\}$.

TSO operational semantics. In the operational semantics we consider an abstract machine executing programs in the core language. Each CPU has a set of general-purpose registers $\mathsf{Reg} = \{\mathtt{r}_1, \ldots, \mathtt{r}_m\}$ storing values from Val. On TSO, processors do not write to memory directly. Instead, every CPU has a ***store buffer***, which holds write requests that were issued by the CPU, but have not yet been ***flushed*** into the shared memory. The state of a buffer is described by a sequence of location-value pairs.

The abstract machine can perform the following transitions:

- $\langle \mathsf{skip} \rangle = \{(\emptyset, \emptyset)\}$;
- $\langle C_1; C_2 \rangle_t = \{(A_1 \uplus A_2, \mathsf{po}_1 \cup \mathsf{po}_2 \cup \{(a,b) \mid a \in A_1 \wedge b \in A_2\})\}$;
- $\langle C_1 + C_2 \rangle_t = \langle C_1 \rangle_t \cup \langle C_2 \rangle_t$;
- $\langle C^* \rangle_t = \{\emptyset, \emptyset\} \cup \{(\biguplus_{i=1}^n A_i, \bigcup_{i=1}^n \mathsf{po}_i \cup \{(a,b) \mid a \in A_i \wedge b \in A_j \wedge i < j\}) \mid (A_i, \mathsf{po}_i) \in \langle C \rangle_t \wedge n \geq 1\}$;
- $\langle m \rangle_t = \{(A \uplus \{c\} \uplus \{d\}, \mathsf{po} \cup \{(c,d)\} \cup \{(c,a),(a,d) \mid a \in A\}) \mid (A, \mathsf{po}) \in \langle C_m \rangle_t \wedge c = (\_, t, \mathsf{call}\ m(\_)) \wedge d = (\_, t, \mathsf{ret}\ m(\_))\}$;
- $\langle \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n \rangle = \mathsf{prefix}(\{(\biguplus_{t=1}^n A_t, \bigcup_{t=1}^n \mathsf{po}_t) \mid (A_t, \mathsf{po}_t) \in \langle C_t \rangle_t,\ t = 1..n\})$, where $\mathsf{prefix}(A, \mathsf{po}) = \bigcup_{a \in A}(\{a' \mid a' \in A \wedge a' \xrightarrow{\mathsf{po}} a\}, \mathsf{po}) \cup \{(A, \mathsf{po})\}$.

Figure 1: Program order semantics of common commands

- A CPU wishing to write a value to a memory location adds an appropriate entry to the *tail* of its store buffer.
- The entry at the *head* of the store buffer of a CPU is flushed into the memory at a non-deterministically chosen time. Store buffers thus have the FIFO ordering.
- A CPU can execute a *memory fence* that flushes all the content of its store buffer to the memory in the FIFO ordering.
- A CPU wishing to read from a memory location first looks it up in its store buffer. If there are entries for this location, it reads the value from the newest one; otherwise, it reads the value directly from the memory.
- A CPU can execute a command affecting only its registers. In particular, it can call a library method or return from it.

3 The TSO axiomatic memory model

Action structures. We record information about program executions using ***actions***, defined as follows:

$$a \in \mathsf{Act} ::= (e,t,\mathsf{store}(x,v)) \mid (e,t,\mathsf{load}(x,v)) \mid (e,t,\mathsf{call}\ m(v)) \mid (e,t,\mathsf{ret}\ m(v)) \mid (e,t,\mathsf{fence})$$

Here $t \in \mathsf{ThreadID}$, $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and e is an *action identifier*, picked from the set AId. For call and return actions v means actual parameter and return value respectively. We omit e annotation from actions, when it is not relevant, and often use r, w and f to denote load, store and fence actions.

We denote the set of all finite sets of actions with $\mathscr{P}(\mathsf{Act})$. When considering a relation R over actions, we write $(a,b) \in \mathsf{R}$ and $a \xrightarrow{\mathsf{R}} b$ interchangeably.

Program order semantics. We first define a *program order semantics*, which generates all executions of a program based solely on the structure of its statements, without taking into account the semantics of memory operations. For instance, in generated executions loads can read arbitrary values disregarding the values written by performed store actions. After we define the notion of execution, we introduce axioms which filter out executions that do not satisfy them.

Program order semantics associates a program with a set of ***action structures***—tuples (A, po), where $A \in \mathscr{P}(\mathsf{Act})$, and $\mathsf{po} : A \times A$ is ***program order*** that is a total on the set of actions by the same thread.

MOWF. mo is total, transitive, irreflexive and relates only store actions in the execution.

POWF. po is total, transitive, irreflexive and relates only actions by the same thread.

SCWF. sc is total, transitive, irreflexive and relates only fences in the execution.

RFWF. $(\forall w_1, w_2, r.\, w_1 \xrightarrow{\mathsf{rf}} r \wedge w_2 \xrightarrow{\mathsf{rf}} r \implies w_1 = w_2) \wedge$
$(\forall w, r.\, w \xrightarrow{\mathsf{rf}} r \implies \exists x, a.\, w = (\_, \mathsf{store}(x,a)) \wedge (r = (\_, \mathsf{load}(x,a))) \vee r = (\_, \mathsf{call}\ \_(a))) \vee r = (\_, \mathsf{ret}\ \_(a))))$

RFDET. $\forall x, w, r.(r = (\_, \mathsf{load}(x, \_)) \vee (\exists t.x = \mathsf{param}_t \wedge r = (t, \mathsf{call}\ \_)) \vee$
$(\exists t.x = \mathsf{retval}_t \wedge r = (t, \mathsf{ret}\ \_))) \wedge w = (\_, \mathsf{store}(x, \_)) \wedge (w \xrightarrow{\mathsf{hb}} r \vee w \xrightarrow{\mathsf{po}} r) \implies \exists w'.w' \xrightarrow{\mathsf{rf}} r$

HBDEF. $\mathsf{hb} = (\mathsf{po} \cup \mathsf{rf})^+$

HBVSMO. $\neg\exists w_1, w_2.$ $w_1 \underset{\mathsf{mo}}{\overset{\mathsf{hb}}{\rightleftarrows}} w_2$

RFMR. $\neg\exists w_1, w_2, r.$ $w_1 \xrightarrow{\mathsf{mo}} w_2 \xrightarrow{\mathsf{hb}} r$, with $w_1 \xrightarrow{\mathsf{rf}} r$

where w_1, w_2 and r access the same location.

RFMR'. $\neg\exists w, w_1, w_2, r.$

$w_1 \xrightarrow{\mathsf{mo}} w_2 \xrightarrow{\mathsf{mo}} w \xrightarrow{\mathsf{rf}} r' \xrightarrow{\mathsf{sb}} r$, with $w_1 \xrightarrow{\mathsf{rf}} r$

where w_1 and w_2 write to the same location, and w and r' are by different threads.

HBWF. hb is acyclic.

HBVSSC. $\neg\exists f_1, f_2.$ $f_1 \underset{\mathsf{sc}}{\overset{\mathsf{hb}}{\rightleftarrows}} f_2$

MOVSSC. $\neg\exists w_1, w_2, f_1, f_2.$

$w_1 \xrightarrow{\mathsf{hb}} f_1 \xrightarrow{\mathsf{sc}} f_2 \xrightarrow{\mathsf{hb}} w_2$, with $w_2 \xrightarrow{\mathsf{mo}} w_1$

SCRF. $\neg\exists w, w', f_1, f_2, r.$

$w \xrightarrow{\mathsf{mo}} w' \xrightarrow{\mathsf{po}} f_1 \xrightarrow{\mathsf{sc}} f_2 \xrightarrow{\mathsf{po}} r$, with $w \xrightarrow{\mathsf{rf}} r$

Figure 2: The validity axioms

Let $A \cup B$ be the union of the sets of actions A and B with disjoint sets of action identifiers. Consider a program $C(L) = (\mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n)$. We define the set of action structures $\langle C(L) \rangle$ for program executions in Figure 1. By construction and definition of $\mathsf{prefix}(A, \mathsf{po})$, $\langle C(L) \rangle$ is also prefix-closed, i.e. it includes incomplete executions.

For a primitive command $\alpha \in \mathsf{PComm}$, we assume a set $\langle \alpha \rangle_t$ of all action structures α produces when executed by thread t. We require that structures in $\langle \alpha \rangle_t$ do not contain call or return actions.

The memory model gives a semantics to a program as a set of ***executions***, each of which is a tuple $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ of a set of actions $A \in \mathscr{P}(\mathsf{Act})$ and relations on A. An execution enriches an action structure with information about the way operations on memory are performed. The relations included into an execution are as follows:

- rf: ***reads-from***, relating the load actions r to the store actions w from which they take their values;
- mo: ***modification order***, relates all store actions in the order they hit the memory;
- sc: ***synchronisation order***, relates all fences in the order of their execution;
- hb: ***happens-before***, showing the precedence of actions in the execution.

Validity. For an execution X and one of the relations R defined above, we write $\mathsf{R}(X)$ to select the corresponding relation for X. An execution $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ is called ***valid*** when it satisfies the validity axioms in Figure 2.

A call action $(t, \mathsf{call}\ m(v))$ gets its argument v by reading from a correspondent client's store to param_t. A return action $(t, \mathsf{ret}\ m(v))$ gets its return value v by reading from a correspondent library's

store to retval_t. In the following we treat calls and returns as loads, what is established in RFWF axiom. We also assume that call (return) actions access $param_t$ ($retval_t$) only when executed by a thread t.

A store action $(t, \mathsf{store}(x,v))$ writes a value v to the address in memory. Analogously to the operational semantics, where a written value does not hit the main memory immediately, in our model a store $(t, \mathsf{store}(x,v))$ becomes observable from a thread t, but not always from the other ones. The explicit way to make the written value visible to the other threads is to execute a fence action (t, fence). We add RFMR, RFMR' and SCRF axioms to ensure that a load action reads the most recent value that is observable to its thread.

We further use $[\![C(L)]\!]$ to denote for a given program $C(L)$ the set of all valid executions with action structures from $\langle C(L)\rangle$: $[\![C(L)]\!] = \{X \mid X = (A, \mathsf{po}, \_, \_, \_, \_) \wedge (A, \mathsf{po}) \in \langle C(L)\rangle\}$.

Execution projections. We call actions of the form $(t, \mathsf{call}\ m(v))$ or $(t, \mathsf{ret}\ m(v))$ ***interface actions***.

Consider an execution $X = (A, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc}, \mathsf{hb})$ of $C(L)$. An action $a \in A$ is a ***library action***, if it is an interface action, or $\exists b.\ b = (\_, \mathsf{call}\ \_) \wedge b \xrightarrow{\mathsf{po}} a \wedge \neg\exists c.\ c = (\_, \mathsf{ret}\ \_) \wedge b \xrightarrow{\mathsf{po}} c \xrightarrow{\mathsf{po}} a$.

An action $a \in A$ is a ***client action***, if it is an interface action, or the negation of the above property holds. Let $\mathsf{client}(A)$ be the set of client actions in A. We define a client's execution:

$$\mathsf{client}(X) = (\mathsf{client}(A), \mathsf{client}(\mathsf{po}), \mathsf{client}(\mathsf{rf}), \mathsf{client}(\mathsf{mo}), \mathsf{client}(\mathsf{hb}))$$

by projecting all the relations in X to actions from $\mathsf{client}(A)$. We also use analogous projection $\mathsf{lib}(X)$ to library actions and lift client and lib to sets of executions pointwise.

Non-interference. We assume that the set of memory locations Loc is partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc}$. The client C and the library L are *non-interfering* in $C(L)$, if in every computation from $[\![C(L)]\!]$, commands performed by the client (library) code access only locations from CLoc (LLoc). Formally, an execution $C(L)$ is called ***non-interfering*** when it satisfies the following axiom:

$$\begin{aligned}&\text{NONINTERF.} \forall a, x, t.\, a \in A \wedge (a = (t, \mathsf{store}(x, \_)) \vee a = (t, \mathsf{load}(x, \_))) \implies \\ &\quad ((a \in \mathsf{lib}(A, \mathsf{po}) \iff (x \in \mathsf{LLoc} \vee a = (t, \mathsf{store}(\mathsf{retval}_t, \_)) \vee a = (t, \mathsf{load}(\mathsf{param}_t, \_)))) \wedge \\ &\qquad (a \in \mathsf{client}(A, \mathsf{po}) \iff (x \in \mathsf{CLoc} \vee a = (t, \mathsf{store}(\mathsf{param}_t, \_)) \vee a = (t, \mathsf{load}(\mathsf{retval}_t, \_))))).\end{aligned}$$

In the following, we assume that at the beginning of execution all locations are arbitrarily and explicitly initialised by means of store actions.

4 Abstraction theorem

The idea behind linearizability is to record all interactions between the client and the library. That is done by means of the notion of a ***history***. Clients and libraries can affect each other by passing different values through interface actions. Precisely, the library can observe the parameters provided by the client at calls, and the client can observe the library's return values. Therefore, a history includes the set of interface actions in the execution. We also include in histories two partial orders (*guarantee* and *deny*) over interface actions to consider additional interactions caused by relaxations of TSO.

Definition 1. A **history** is a set of interface actions and a pair of partial orders over it.

Informally, in a history $H = (I, G, D)$, the ***guarantee*** G describes the happens-before edges enforced by the library; and the ***deny*** D describes the happens-before edges that a client must not enforce.

Consider an execution X and its interface actions $I(X)$. We let $\mathsf{guar}(X)$ be the projection of $\mathsf{hb}(X)$ onto $I(X)$ and $\mathsf{deny}(X)$ be the relation over $I(X)$ obtained from dashed edges in Figure 3. These edges

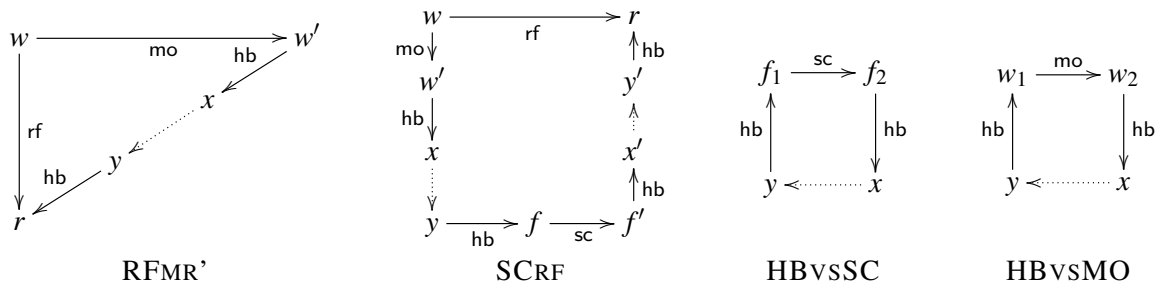


Figure 3: The definition of the $\mathsf{deny}(X)$. Here x, y, x', y' are interface actions. If the solid edges belong to the corresponding relations in X, the dashed edges belong to $\mathsf{deny}(X)$.

describe all the possible ways in which happens-before edges enforced by the client can contradict other relations from the library execution; the axioms that can be violated are indicated in the figure. Let $\mathsf{history}(X) = (I(X), \mathsf{guar}(X), \mathsf{deny}(X))$. We lift history to sets of executions pointwise.

We use the $\subseteq$ relation between partial orders to denote that a partial order is a sub-relation of another one, and lift it to histories as follows: $(I_1, G_1, D_1) \subseteq (I_2, G_2, D_2) = (I_1 = I_2) \wedge (G_1 \subseteq G_2) \wedge (D_1 \subseteq D_2)$.

Definition 2. A history H' **linearizes** a history H if $H' \subseteq H$.

Thus, a linearized history enforces fewer dependencies between interface actions for a client and has less restrictions on enforcing dependencies by a client. This indeed allows more client behaviours.

To generate the set of all histories of a given library L, we consider its *most general client*, whose threads repeatedly invoke library methods in any order and with any parameters possible. Take $n \geq 1$ and assume $\mathsf{sig}(L) = \{m_1, \ldots, m_l\}$. Then we define $\mathsf{MGC}_n(L) = (\mathsf{let}\ L\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_n^{\mathsf{mgc}})$, where for all t, $C_t^{\mathsf{mgc}} = (m_1 + \ldots + m_l)^*$.

A ***library execution*** of L is an execution of $[\![\mathsf{MGC}_n(L)]\!]$ for some $n \geq 1$. A library execution is ***valid***, if it satisfies the validity axioms in Figure 2, and it is ***non-interfering***, if it satisfies NONINTERF axiom.

Let $[\![L]\!]$ be the set of all valid library executions of L. The set of executions in $[\![L]\!]$ defines a *library-local semantics* of L. We say that a library L is non-interfering if so is every execution in $[\![L]\!]$.

Now we present our main result – the definition of linearizability for the axiomatic model of TSO. The correctness of the proposed notion is established in Theorem 4.

Definition 3. For non-interfering libraries L_1 and L_2, L_2 **linearizes** L_1, written $L_1 \sqsubseteq L_2$, if:

$$\forall H_1 \in \mathsf{history}([\![L_1]\!]).\, \exists H_2 \in \mathsf{history}([\![L_2]\!]).\, H_2 \subseteq H_1.$$

Noteworthily, checking linearizability $L_1 \sqsubseteq L_2$ does not involve reasoning about any client. What we need to do is to generate all possible library-local executions, or, in other words, all possible executions of $\mathsf{MGC}(L_1)$ and $\mathsf{MGC}(L_2)$, and check the definition.

Theorem 4 (Abstraction)**.** *If L_1, L_2 and $C(L_2)$ are non-interfering and $L_1 \sqsubseteq L_2$, then $C(L_1)$ is non-interfering and* $\forall X_1, X_2\ .\ X_1 \in \mathsf{client}([\![C(L_1)]\!]) \wedge X_2 \in \mathsf{client}([\![C(L_2)]\!]) \wedge \mathsf{hb}(X_2) \subseteq \mathsf{hb}(X_1)$.

By Theorem 4, while reasoning about a client $C(L_1)$ of a library L_1, we can soundly replace L_1 with a simpler library L_2 linearizing L_1: if a property over client actions holds over $C(L_2)$, it will also hold over $C(L_1)$. Since L_2 is usually simpler than L_1, this eases the proof of the resulting program. Thus, the proposed notion of linearizability and Theorem 4 enable compositional reasoning about programs on TSO: they allow decomposing the verification of a whole program into the verification of its constituent components.

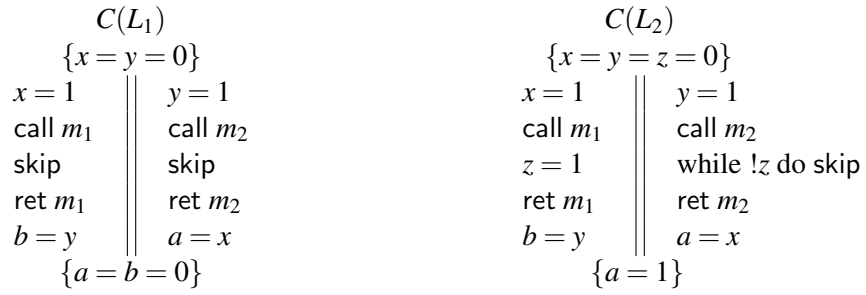


Figure 4: A motivation to include guarantee: it should not be the case that $L_1 \sqsubseteq L_2$, if the latter adds synchronisation and therefore allows less behaviours.

Figure 5: A motivation to include deny: if $L_1 \sqsubseteq L_2$, then client must not be able to distinguish them by making a synchronisation between library methods calls.

Let us now return to definition of a history and illustrate the ideas behind inclusion of two relations into it. The inclusion of a *guarantee* relation into a history is aimed to prevent a library L_2 that linearizes L_1 from adding new synchronisations, so that a client cannot notice a difference between them.

Consider libraries L_1 and L_2 with different implementations for methods m_1 and m_2 along with their client C in Figure 4. A client is able to observe a synchronisation inside a library L_2, because it disables some client's behaviours. Particularly, because of reading z in m_2, a store $x = 1$ is always flushed by the moment of reading $a = x$. Consequently, the outcome $b = 0$ is possible in $C(L_1)$ and never happens in $C(L_2)$. In our setting such case is ruled out by a guarantee relation, since all histories of L_2 contain an edge (between a call to m_1 and a return from m_2) that any history of L_1 does not.

With the following example we show the role of a *deny* relation in a history. Consider libraries L_1 and L_2 with different implementations for a method m_2 along with their client in Figure 5. The former one always returns 2 while the latter returns the value of $x{+}{+}$. It is easy to see that for any history from $\mathsf{history}([\![L_1]\!])$ there is an equivalent one from $\mathsf{history}([\![L_2]\!])$, so by definition $L_1 \sqsubseteq L_2$. However, Abstraction Theorem does not hold of L_1 and L_2. The subtlety here is that a client is able to perform a synchronisation that influences library's execution and makes it possible to detect a different behaviour of a linearized library.

In terms of our axiomatic model this means, that an execution of $C(L_2)$ violates RFMR' validity axiom, while $C(L_1)$ does not. To avoid this, each validity axiom that can be violated because of client's synchronisation contributes edges into a *deny* relation. This way any client synchronisation that breaks a library-local execution is explicitly forbidden.

5 Related work and conclusions

Recent work has proposed definitions of linearizability for the operational model of TSO [2] and the axiomatic model of C++11 [1]; the latter memory model is significantly more complex than TSO. The

techniques we used in this paper are inspired by the construction of the definition of linearizability for C++11. By demonstrating their application in a simple and clean setting, we hope to highlight their main underlying ideas and make it easier for other researchers to use them for developing compositional reasoning methods for other memory models.

We also hope that the definition of linearizability for an axiomatic version of TSO will lend itself easier to automatic verification and testing than the operational definition [2]. Namely, model checking the latter requires enumerating an exponential number of concurrent program executions. In contrast, a single execution in an axiomatic model concisely represents whole classes of executions in a way that can be accepted by standard SAT or SMT solvers, which enables efficient verification [4].

References

[1] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. POPL, 2013. To appear.

[2] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 1990.

[4] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[5] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[6] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

Hybrid type systems

Jose A. Lopes
Max Planck Institute for Software Systems
Saarbrücken, Germany
jose@mpi-sws.org
(advised by Ruzica Piskac)

Abstract

Hybrid type systems combine the best of static and dynamic type systems, providing an adequate compromise when properties are at conflict. Moreover, the combination of gradual typing with hybrid type checking gives users control over the level of typedness. As a result, users can adjust the type system to the particular needs of each stage in the development process and avoid the scenario where the type system becomes a burden. In this paper, we review existing approaches to hybrid type systems from which we collect a number of relevant properties, such as, static detection of type errors and type inference. Finally, we present these properties and discuss our future research directions.

1 Introduction

Increasing software reliability is one of the main tasks of researchers and engineers working in program verification and software engineering. There are various powerful formal methods for proving correctness properties of programs. However, in order to fully explore their potential, users must have a high level of expertise and a deep understanding of the underlying theories. An average programmer is more inclined towards methods that help increase software reliability and correctness without being exposed to a complicated formalism. Such methods are known as *lightweight formal methods* and type systems are perhaps the most widely used among them.

A type system is a mechanism in a programming language that ensures a certain level of correctness and that can specify and verify basic interface specifications [5]. Additionally, by using type declarations, programs are more structured, and thus amenable to higher level verification techniques. Traditionally, type systems are divided in static and dynamic type systems. In static type systems, type errors are caught at compile-time and ill-typed programs cannot be executed. On the other hand, dynamic type systems detect type errors at runtime and programs are aborted [4]. In general, properties supported by one system are not supported by the other.

Static type systems allow earlier error detection, better documentation through type signatures, compiler optimization, and increased runtime efficiency [1]. However, contrary to advocates of static typing, well-typed programs can still go wrong, mainly because static type checking is partially sound and incomplete. Additionally, static type systems enforce a tight typing discipline making a language less expressive. Due to loss of expressiveness and generality, some classes of well-typed programs do not type check in conventional static type systems. Among the examples for the programs that do not type check under conventional static type systems are heterogeneous functions (functions returning non-uniform types) and some forms of recursion (for example, self application results in an infinite type) [4].

Dynamic type systems address these problems and among their advantages are truly dynamic behavior, rapid development, fast adaptation to changing or unknown requirements, and simpler interaction

with systems that change unpredictably [1]. Moreover, users are not forced to commit prematurely to a particular data representation and they can experiment more easily. While dynamic languages, such as Python and Ruby, allow several forms of reflection and intercession (the ability to inspect and modify the program at runtime), these properties are usually not supported (in full) by static languages. However, their use makes code more difficult to read and understand, and program errors can be very difficult to trace. In addition, the advantages of the static languages are mainly disadvantages for the dynamic languages: they do not support the standard stronghold of static types, such as earlier error detection, documentation, optimization, and runtime efficiency.

Such a division between static and dynamic type systems forces users to make a choice that is not always obvious because some useful properties are always left out. Hybrid type systems appear as a solution to this problem, by combining the best of both static and dynamic type systems. Hybrid type systems are also a topic on which I am planning to work during my PhD studies.

I have started a PhD program only recently (one month ago), and this paper is a survey where we present current work done towards defining hybrid type systems and show the advantages and limitations of each attempt.

2 Problem statement

Our goal is to develop a hybrid type system that combines a number of relevant and useful properties, from static and dynamic type systems, and implements them efficiently, with an adequate level of compromise when needed.

We plan to develop a gradual hybrid type system. Following [2], graduality is an important requirement that allows users to control the degree of typedness. With this approach users can adjust the type system to the particular needs of each stage of development. As an illustration, consider the following scenario: at the start of a software project, users are still sketching the program and, therefore, the type system should not enforce a strong typing discipline. This gives users more freedom in experimenting with different models. When the program becomes more mature, users can add type annotations, which can be used by the type system to catch more type errors statically and generate optimized code.

By developing a hybrid type system, we also aim to increase software reliability. A division of static and dynamic type systems also extends to a division among verification techniques: specifications can be verified either statically or dynamically [5]. Hybrid type systems provide an ideal infrastructure to support these two classes of specifications in the same programming language. Once again, users do not have to choose and we plan to combine static and dynamic type systems and specifications.

We see this project as part of a bigger research question that answers how type systems can evolve throughout the development process to adjust to the particular needs at each stage. Our study of current approaches to hybrid type systems has showed that these type systems have great potential for programming languages by providing a unified static/dynamic framework for specification and verification.

In their position paper, Meijer and Drayton [1] proposed a number of relevant properties a hybrid type system should provide. However, the approaches we have studied (cf. Section 3) showed that only a very small subset of properties are considered for a given type system and that the level of compromise is extreme. As an illustration, a hybrid type system [4] supports dynamically typed programming but it does not catch any type errors statically. In our opinion, such restrictions are too drastic and a compromise should be found instead. We are aware that it is impossible to meet all the requirements, as there are several degrees of conflict. However, we plan to explore the fact that at a particular stage in the development process not all properties need to be in effect. Therefore, the type system can evolve from

a more dynamic to a more static one.

There are several open questions regarding hybrid type systems, among the most important ones is the right set of properties and an adequate level of compromise such that the type system is feasible and useful. Our ongoing research work aims at answering these questions. In Section 4 we list desired relevant properties for a hybrid type system. This list is based on the extensive study of related work.

3 Related work - Approaches to hybrid type systems

This section describes several approaches towards hybrid type systems. We list their main contributions, as well as their limitations.

3.1 Static typing with dynamic type

Abadi et al. [3] proposed an approach to hybrid type systems that comprises a static type system, defined by a typed lambda-calculus, with a dynamic type denoted by *Dynamic*. Dynamic values can be created with the type constructor *dynamic* and inspected with the *typecase* form. A dynamic value is a pair (v,T) where v is a value and T is a type tag containing the type of v. For example, the following higher-order function receives two dynamic values *df* and *de*, and uses the *typecase* form to dynamically ensure that *df* has function type and that *de* matches the type of the function argument.

```
λdf: Dynamic. λde: Dynamic.
  typecase df of
    (X, Y) (f: X → Y)
    typecase de of
      (e. X) dynamic f(e): Y
      else dynamic "Error: formal and actual types mismatch" : String
    end
  else dynamic "Error: expected function type" : String end
```

In general, this approach to hybrid type systems makes the type system more flexible, when compared to a conventional static type system. However, it still does not allow programming in a dynamically typed style because the programmer is forced to insert coercions from and to type *dynamic* [2]. Moreover, there is a too fine line separating the highlevel semantics and the underlying representation of dynamic values. Therefore, users must not only manage the interface between static and dynamic types manually, but also remember the history of tagging patterns of a value because a runtime check must look for the exact same pattern [6]. The hybrid type system explained in the next section addresses this issue.

3.2 Quasi-static typing

Thatte [6] introduces an approach to hybrid type checking called quasi-static typing. In this approach, programs are classified as ill-typed, programs that are rejected because they contain one or more statically detected type errors, well-typed, programs that are accepted because they do not contain static type errors or produce unexpected runtime type errors, and ambivalent, programs that are accepted because they do not contain any static type errors, although they may contain runtime type errors.

In quasi-static typing, dynamic typing is a property inherited by all types through structural subtyping. The type of dynamically typed values is Ω and all types are subtypes of it. Quasi-static typing has two phases, namely, type and coercion inference, and plausibility checking. Type and coercion inference

reconstruct types for terms and insert coercions in ambivalent programs, making them well-typed. Plausibility checking detects type errors statically by reducing sequences of tagging and checking operations and locating implausible coercions, such as, a coercion from *Nat* to *Bool*.

Quasi-static typing merges static and dynamic typing, which combined with type inference and implicit coercions provide a good user experience. However, there are some limitations. For example, this approach does not statically catch all type errors in fully annotated programs, and the type hierarchy with the type dynamic as the generalization of all other types combined with negative coercions weakens the type system up to a point that too many ill-typed programs are accepted [2]. Cartwright et al. [4] point out other limitations, namely, function arguments require type annotations, only one level of subtyping is allowed, and parametric polymorphism and recursive types are not provided.

Because parametric polymorphism is not supported, there is an over generalization of types. For example, the polymorphic identity function has type $a \to a$, thus preserving the argument type in the result. However, in quasi-static typing, the identity function has type $\Omega \to \Omega$, therefore, arguments are coerced to type dynamic. As a result, the type of the argument cannot be propagated, which would be useful to statically detect more type errors. Moreover, as shown in the next section, recursive types are a possible solution to the heterogeneity problem, e.g., a function returning different types.

3.3 Soft typing

Cartwright et al. [4] propose an approach to hybrid type systems partly based on quasi-static typing. Soft typing classifies programs as well-typed and ambivalent only, and ill-typed programs are considered ambivalent. The type system inserts runtime checks in ambivalent programs around suspect arguments of primitive operations, thus converting dynamically typed programs into well-typed programs. The inserted runtime checks are located in expressions that might contain type errors, but type errors are not caught statically [2]. As a result, users must inspect program phrases that contain runtime checks for potential type errors. Soft typing provides several type features, namely, parametric polymorphism, type variables, type constructors, union types, and recursive types (through fixed-point operations on type functions). Union types can be used to encode the type of a conditional that produces different return types on its branches. For example, the following function has type $true + false \to suc + nil$, where *suc* is a constructor for natural types, and *nil* a constructor for the empty list.

```
λ x. if x then 1 else nil
```

Contrary to static typing with dynamic type, in soft typing the user does need to explicitly annotate dynamically typed variables or resort to checking operations on dynamic values, to ensure a correct static typing discipline or to manage the interface between statically typed and dynamically values.

3.4 Gradual typing

Siek et al. [2] points out that soft typing does not catch errors statically. As a result, a different approach to hybrid type system called gradual typing is proposed. Being a gradual type system, it gives the user control over the degree of static checking by optionally annotating function parameters with types. The proposed gradual type system supports dynamically typed programming when type annotations are omitted by the user and statically typed programming otherwise. The benefits of static typing include static error detection, optimized code generation (e.g., unboxed values), and improved runtime efficiency, by avoiding unnecessary runtime checks. It should be noted that this system catches all type errors statically for fully annotated programs, a property that is not provided by the previous hybrid type systems.

The proposed type system is a gradually typed λ-calculus, which is an extension to the simply typed λ-calculus with the dynamic type, denoted by *?*. The gradually typed λ-calculus is then transformed into a simply typed λ-calculus with explicit casts. Runtime checks are also necessary because unannotated type parameters can prevent static detection of some type errors. For example, the parameter x in the following function has type dynamic.

```
((λ (x) (succ x)) #t)
```

Therefore, even though the program is ill-typed, this error is caught only at runtime by the inserted checks. In this case, type inference could reconstruct the type of variable x and detect type inconsistency statically. However, type inference is not provided. Another interesting property would be parametric polymorphism, but it is also not supported. As a result, the gradual type system suffers from the same problem of quasi-static typing, in what overgeneralization of types is concerned (Section 3.2).

3.5 Hybrid type checking

The hybrid type system approaches presented so far have mainly focused on type checking and inferencing, and language features, such as parametric polymorphism, and union and recursive types. This section presents a different approach to hybrid type systems based on verification of specifications.

Flanagan [5] proposed a hybrid type checking approach that combines traditional static type systems for verifying statically basic interface specifications, whenever possible, and dynamic contract checking to support more expressive and precise specifications.

In general, the kinds of specifications provided by static type systems are limited. Dependent and refinement types appear in these type systems as more expressive specifications. But even these mechanisms are not expressive enough because the specification language is intently restricted to preserve static verification. On the other hand, dynamic checking supports more precise specifications, such as, subrange types, aliasing restrictions, ordering restrictions, size specifications, and ultimately any arbitrary predicate, with the disadvantage of not being statically verifiable in general.

The underlying infrastructure of Flanagan's hybrid type checking is similar to that of quasi-static typing, soft typing, and gradual typing: programs are classified as well-typed, ill-typed, or ambivalent, and, in the latter case, runtime checks and coercions are inserted to detect type errors at runtime. The main difference is that dependent function types, refinement types, and arbitrary predicates, are also used, allowing users to define precise and expressive specifications.

The refinement types are written using constants, namely, boolean and arithmetic values and operations, conditionals, and fixpoint constructors. The type system relies on an automatic theorem prover to statically verify the specifications, but the use of arbitrary predicates makes typechecking undecidable. In this case, the program is still accepted and the hybrid type checker inserts casts and runtime checks to verify the specification dynamically.

4 Properties of hybrid type systems

In the previous section, we showed several approaches to hybrid type systems, each emphasizing different properties, in some cases preferring static typing over dynamic or vice-versa. Inspired by these type systems and [1], we propose a set of properties that a hybrid type system should provide. To our knowledge, no current hybrid type system supports all of these properties.

Minimal text principle Type annotations in terms are optional. However, unannotated terms do not default to the dynamic type. Instead, type inference is used and parametric polymorphism allowed.

Static error detection There is a best-effort to statically detect type errors. However, all type errors are caught in fully annotated programs.

Type inference There is a best effort to reconstruct the types of terms, while at the same time avoiding the overgeneralization problem described in Section 3.2

Gradual typing A gradual transition between dynamic and static typing is provided, with the user controlling the typedness degree through type annotations.

Specifications Specifications are provided to allow definition of invariants. However dynamically verifiable specifications are not forbidden and are deferred to runtime, and there is no syntactic distinction between these and statically verifiable specifications.

Subtyping In the presence of inheritance, when there is a subtype relationship between the inferred and required types, for example, in a function application, coercions are implicitly inserted.

Generics and heterogeneous data Data types are parametric to promote code reuse. Moreover, heterogeneous data structures are allowed because, with homogeneous structures only, users are forced to commit (prematurely) to a particular data representation.

Covariance Without covariance, parametric types are achieved with parametric polymorphism, causing type parameters to spread. Covariance can be achieved without complicating the type system by adding runtime checks. For example, array covariance combines parametric and subtype polymorphism and each write operation requires a runtime check to preserve the array type.

Prototype programming It should be possible to extend classes and data structures dynamically to avoid the problem that statically typed languages force programmers to commit (prematurely) to inter-entity relationships.

5 Conclusions

To conclude, hybrid type systems are a research area with a great potential. They provide a general unification of both static and dynamic type systems, and statically and dynamically verifiable specifications. This way users are not forced to compromise between static and dynamic type systems and they do not need to commit to a particular type system during an entire development process.

Our research plan is to build such a hybrid type system, that meets the before mentioned properties.

References

[1] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[2] J. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. In *ACM Transactions on Programming Languages and Systems*, pages 237–268, 13(2), 1991.

[4] R. Cartwright and M. Fagan. Soft typing. In *PLDI'91*. 1991. ACM Press.

[5] C. Flanagan Hybrid type checking. In *POPL '06: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pa es 245–256, Charleston, South Carolina, January 2006.

[6] S. Thatte. Quasi-static typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.

Formal Verification of Mobile Robot Protocols

Laure Millet
UPMC Sorbonne Universités,
LIP6-CNRS 7606
Laure.Millet@lip6.fr

Abstract

Mobile robot networks emerged in the past few years as a promising distributed computing model, where simple limited entities perform complex collaborative tasks. Existing works in the literature verify mobile robot protocols by writing *ad hoc* proofs, which, in the case of more complex asynchronous environments, reveals both cumbersome and error-prone.

In this paper, we propose the first formal modeling and verification methodology for mobile robots protocols operating in a discrete space. We use a known protocol for exclusive perpetual exploration with three robots as a case study. Our technique shows that in the synchronous and semi-synchronous environments, this protocol is correct, yet safety properties can be violated in the completely asynchronous environment. Finally, we present a protocol fix for the safety problem and verify the correctness of our change.

1 Introduction

The number of tasks that may be performed by autonomous robots and their complexity are increasing. Many applications envision groups of mobile robots self organizing and cooperating toward the resolution of common objectives, in the absence of any coordinating agent.

A recent trend was to shift from the classical continuous setting where coordinated distributed robots evolve in a continuous two-dimensional Euclidian space, to a discrete one where space is partitioned into a *finite* number of locations. This discrete space is conveniently represented by a graph, where nodes represent locations, and edges represent paths for a robot from one location to another. Thus, the discrete setting permits to simplify robot models by reasoning on finite structures.

We consider a distributed system of k mobile robots, that have limited capacities: they are identical and anonymous (they execute the same algorithm and they cannot be distinguished using their appearance), they are oblivious (they have no memory of their past actions) and they have neither a common sense of direction, nor a common handedness (chirality). Furthermore robots do not communicate in an explicit way. However they have the ability to sense the environment and see the position of the other robots. Robots operate in three phase cycles: *Look*, *Compute* and *Move*. During the *Look* phase robots take a snapshot of the graph together with other robots' positions. The collected information is used in the *Compute* phase in which robots decide to move or to stay idle. In the *Move* phase, robots may move to one of their adjacent nodes computed in the previous phase. In this model introduced by Suzuki & Yamashita in [7] called SYm (or ATOM), a subset of robots execute the three phases synchronously. Prencipe improved this model by proposing in [6] the CORDA model which is totally asynchronous, and reflects distributed system behavior. We consider execution models where movements are instantaneous.

The research concern is on determining what tasks can be performed, under what conditions, and at what cost. Among the multiple tasks that have been studied in a discrete setting, we study here

the perpetual exploration of an unknown environment: Every node of the graph which constitute the environment must be explored infinitely often. We assume in this work that the graph shape is a ring.

No deterministic exploration is possible on a ring when the number of robots k divides the number of nodes n [4]. The authors proposed a deterministic algorithm to solve the exploration using at least 17 robots provided that n and k are co-prime. In [3] the authors show that four identical probabilistic robots are necessary and sufficient in any anonymous unoriented ring of size $n > 8$, also removing the co-prime constraint and propose a probabilistic algorithm for four robots in any ring of size $n \leq 8$. In [2], the authors investigate both the minimal and the maximal number of robots that are necessary and sufficient. On the minimal side, three deterministic robots are necessary and sufficient, in a ring of size $n \geq 10$. On the maximal side, $n-5$ robots are necessary and sufficient to exclusively perpetually explore a ring of size n when n is coprime with k. They provide algorithms for both the minimal and maximal case.

For these algorithms, no formal correctness proofs have been given. We propose the first formal modeling and verification methodology for mobile robots protocols. Our case study concerns algorithms proposed in [2] which permit the exclusive perpetual exploration problem on ring shaped graphs.

We first present our formal model, then we discuss the case study specifications and properties which need to be verified. Finally we present verification for algorithms from [2], and the fixed and verified algorithm in the last section.

2 System model

We describe the robot behavior in SYm or CORDA execution models, as a set of finite automata. We consider two types of components, robots and scheduler, each of them described by an automaton. We first discuss robot modeling, before going on to how the total asynchrony or strong atomicity for robot behaviors are performed by the scheduler according to the execution model.

Robot modeling. Robots execute the same deterministic algorithm and have an identical behavior, hence they can be described by the same automaton. Figure 1 shows a finite automaton modeling the robot behavior. It should be recall that robots operate in *Look*, *Compute*, and *Move cycles*.

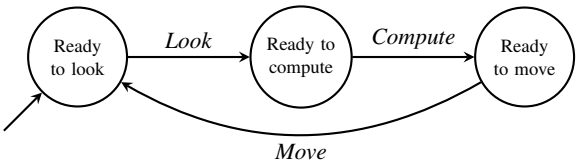


Figure 1: An automaton for the robot behavior

To begin a cycle, a robot must take a snapshot of its environment, which is represented by the *Look* transition. Then, it must compute its future location, represented by the *Compute* transition. Finally the robot has to move according to its previous computation, this effective movement is represented by the *Move* transition.

The "Ready to move" state is divided in as many parts as the number of possible movements according to the algorithm to be verified.

We abstract precise time constraints (like computational or motorial speed of robots) and keep only sequences of instantaneous actions, assuming that each robot completes each cycle in finite time. Therefore, the model can be reduced by combining the *Look* and *Compute* phases to obtain the *LC* phase.

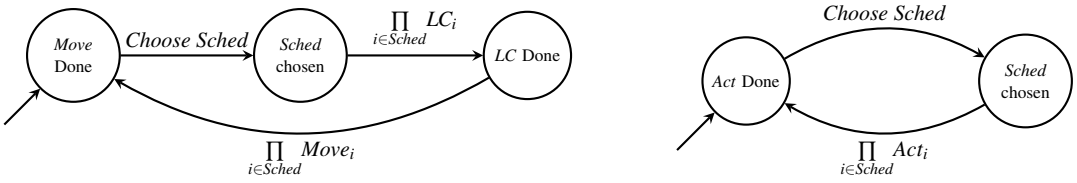
Scheduler modeling. The scheduler organizes robot movements to obtain behaviors which respect SYm and CORDA models. Like robots, the scheduler is modeled by a finite automaton. For each variant of the execution model, there is one scheduler model. By synchronizing one of these schedulers with robot automata, we obtain an automaton that represents the global behavior of robots in the chosen model. We now describe these scheduler models for a set *Rob* of robots. Unlike robots which have the same behavior regardless of the model, the scheduler is set by the model and the number of robots.

In the sequel we denote by LC_i (respectively $Move_i$), the *LC* (resp. *Move*) phase of ith robot. And for a subset $Sched \subseteq Rob$, we denote by $\prod_{i \in Sched} LC_i$ (respectively $\prod_{i \in Sched} Move_i$) the synchronized transitions.

The **SYm** model has two variants, SYm Fully-synchronous (all robots are synchronized on every phase) and SYm Semi-synchronous (a subset of robots are synchronized). In the Semi-synchronous case, the automaton consists of a cycle, where a set "*Sched*" is first chosen, then the *LC* and *Move* phases are synchronized for this set. The automaton of SYm Semi-synchronous is described in Figure 2a.

The "*Sched* chosen" state is divided into 2^k states, where k is the number of robots in order to represent all possible sets of $Sched \subseteq Rob$.

In the SYm fully-synchronous variant, each phase of all robots must be synchronized. On all global cycles $Sched = Rob$, thus all robots are always scheduled and synchronized on every phase. Hence all global cycles are identical.



(a) SYm Semi-synchronous model

(b) CORDA model

Figure 2: The Schedulers automata

The **CORDA** model is totally asynchronous. Any finite delay may elapse between *LC* and *Move* phases: A robot can move according to an outdated observation, and any set $Sched \subseteq Rob$ can be scheduled.

The automaton in Figure 2b represents the corresponding scheduler. In each phase a set *Sched* is chosen, and all robots in this set are allowed to act: the action Act_i is either LC_i or $Move_i$ depending on the current state of the i^{th} robot.

System modelling. In order to obtain the complete system, configurations must be added: A configuration describes the positions of robots on the graph. In a graph of n nodes with k robots there are $\binom{n}{k}$ possible configurations, thus the total number of states is multiplied by $\binom{n}{k}$. Furthermore the number of transitions depends on the number of states and on the number of possible movements, thus to represent the system as an automaton every $Move_i$ transition must be divided according to the shape of the graph.

Even with a high level of abstraction, chosen in order to express only behavioral specifications that permits to verify some desired properties, the system automaton made by synchronization is prone to state space explosion.

3 Case Study and Verification

We choose to verify the *Min-Algorithm* presented by [2]. For this, we must outline the properties that need to be satisfied. These properties come from the problem specification, given below:

The problem specification. We study the *Exclusive Perpetual Exploration* problem in the general asynchronous model defined in [2] as follows:

For any graph G of size n and any initial configuration where robots are located on different vertices, an algorithm solves the perpetual exclusive exploration problem if it guarantees these two properties:

- **exclusivity:** No two robots visit the same vertex or traverse the same edge at the same time. It permits to describe the collision problem that can occur during the exploration: It corresponds to mutual exclusion with respect to the positions in the graph and it is discussed in [5]. It assumes that there will never be more than one robot on any vertex nor two robots will traverse the same edge at the same time.
- **liveness:** Each robot visits each vertex in G infinitely often. The liveness property is satisfied *iff* the perpetual exploration is achieved. It implies that executions are infinite and that in any configuration at least one robot can move.

In our models an execution where no robots is ever scheduled can happen, as well as an execution where a particular robot is never scheduled. To prevent such executions a fairness assumption has to be made: All robots have to be scheduled infinitely often. Thus the *liveness* property is satisfied only on executions where the *fairness* assumption holds.

The Min-Algorithm. In [2] the authors proposed the *Min-Algorithm* which ensures that three robots always exclusively and perpetually explore any ring of size $n \geq 10$ where n is not a multiple of k. This algorithm is based on a classification of configurations:

Definition 1. For k robots in the n-node ring, a configuration is a circular and non oriented alternating sequence of symbols R and F, indexed by integers: R_i stands for i consecutive nodes, each of them occupied by a robot, and F_j stands for j consecutive nodes free of robots.

A closed class of configurations is outline. These configurations, called legitimate configurations, are defined by: $C0 = (R_2, F_2, R_1, F_z)$, $C1 = (R_1, F_1, R_1, F_2, R_1, F_z)$ and $C2 = (R_2, F_3, R_1, F_z)$ with $z \in \{0,1,2,3\}$. The phase occurring on these configurations is called the *Legitimate* phase. When started in a legitimate configuration the protocol always moves into a legitimate configuration, after the execution of n rounds, all robots have explored the ring. When started in a non-legitimate configuration the protocol ensures the convergence towards a legitimate configuration thanks to the convergence phase. The algorithm is correct *iff* from any configuration, it converges to a legitimate configuration The legitimate phase (respectively the convergence phase) can be seen on the Table 1 (resp. Table 2)

Legitimate Phase: $z \neq \{0,1,2,3,4\}$			
*RL*1::	(R_2, F_2, R_1, F_z)	$\rightarrow$	$(R_1, F_1, R_1, F_2, R_1, F_{z-1})$
*RL*2::	$(R_1, F_1, R_1, F_2, R_1, F_z)$	$\rightarrow$	(R_2, F_3, R_1, F_z)
*RL*3::	(R_2, F_3, R_1, F_z)	$\rightarrow$	(R_2, F_2, R_1, F_{z+1})

Table 1: Rules of *Min-Algorithm* legitimate phase

Convergence Phase: Execution starting from special configurations.				
$RC1$::	(R_2,F_y,R_1,F_z)	$\rightarrow$	$(R_2,F_{min(y,z)},R_1,F_{max(y,z)+1})$	with $y \neq z \neq \{1,2,3\}$
$RC2$::	$(R_1,F_x,R_1,F_y,R_1,F_y)$	$\rightarrow$	$(R_1,F_x,R_1,F_{y-1},R_1,F_{y+1})$	with $x \neq y \neq 0$
$RC3$::	$(R_1,F_x,R_1,F_y,R_1,F_z)$	$\rightarrow$	$(R_1,F_{x-1},R_1,F_{y+1},R_1,F_z)$	with $x < y < z$
$RC4$::	(R_3,F_z)	$\rightarrow$	(R_2,F_1,R_1,F_{z-1})	when 1 robot executes
		$\rightarrow$	$(R_1,F_1,R_1,F_1,R_1,F_{z-2})$	when 2 robots execute
$RC5$::	(R_2,F_1,R_1,F_z)	$\rightarrow$	(R_2,F_2,R_1,F_{z-1})	

Table 2: Rules of *Min-Algorithm* convergence phase

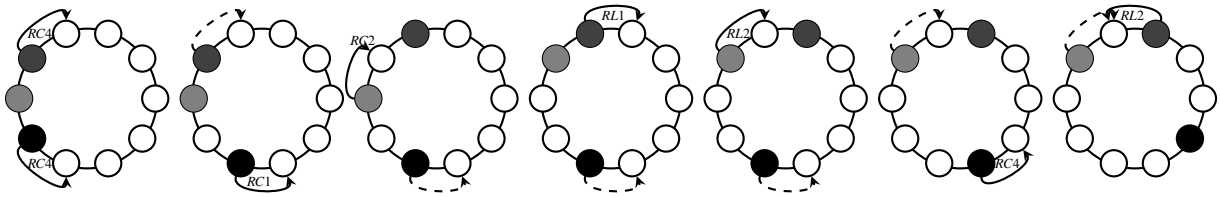


Figure 3: Counter-example

Verification. To represent this algorithm in SYm or CORDA model, guards are added on the actions of robots to constrain their behavior. The *Min-Algorithm* is based on the observed configurations: To every possible configuration, a robot movement is assigned. Thus, each guard is a comparison of the observed configuration and the previously fixed set of configurations.

We use the LTL (Linear Temporal Logic) model-checker DiVinE [1] to check the correctness of the algorithm. This choice is motivated by further use of the input format, which is compatible with other tools developed in our team. The results for the smallest ring, of size 10, are presented in Table 3.

nb States	nb Transitions	Memory (kB)	Model	Verification
256.315	737.810	248.668	SYm Fully-synch	ok
407.175	881.437	248.840	SYm Semi-synch	ok
3.429.715	13.218.742	1.269.432	CORDA	collision

Table 3: Model-checking of *Min-Algorithm* in the three models for the smallest ring

In order to show factors of state space explosion, we outline the number of states, transitions, the memory used, and the time spend. Furthermore the results show that this algorithm does not verify the exclusivity property in the CORDA model. The counter-example of Figure 3 shows an execution which does not satisfied this property. Every ring represents a configuration, a configuration's change occurs when a robot moves, in each configuration a computation is represented by a full arrow. And a computation made on an outdated snapshot by a dotted arrow.

Thus a new algorithm is proposed by the authors of [2]. The legitimate phase is the same, only the convergent phase changes, more precisely, only rule $RC5$ changes to avoid collisions which arose from the previous rules, when movements computed on obsolete observations are taken into account. The new $RC5$ rule is:

$$RC5 \;::\; (R_2,F_1,R_1,F_z) \;\rightarrow\; (R_1,F_1,R_1,F_1,R_1,F_{z-1})$$

We have verified this new algorithm, we obtain the result of Table 4.

n	nb States	nb Transitions	Memory (kB)	Time
10	1.581.961	6.090.209	1.416.880	6min 45s
11	1.926.385	7.421.315	1.568.748	9min 09s
13	2.716.637	10.476.317	2.252.600	20min 46s
14	3.162.409	12.307.905	2.560.724	26min 54s
16	4.155.385	16.041.365	2.772.188	36min 22s

Table 4: Model-checking of the patched *Min-Algorithm*

4 Conclusion

We have proposed a formal modeling and a verification methodology for CORDA and SYm robots models, operating in a discrete space.

As a case study, we chose to verify an algorithm proposed in [2] which permits to achieve exclusive perpetual exploration with three robots in any ring of size $n \geq 10$. We have shown that the algorithm is correct (for $n = 10$) in the fully-synchronous and semi-synchronous environments, but the exclusivity property can be violated in the asynchronous environment. Thus, this algorithm has been fixed in order to avoid such faulty behaviors. The correctness of this fix has been verified, for values of n up to 16. Other techniques should be used to obtain a parametric proof for arbitrary values of n.

Future work will consist in verifying robot protocols, on other graph shapes, in a possibly stochastic settings. Also, instead of verifying a given algorithm, we intend to study the controller's synthesis for the robot models: for this more difficult problem, the aim is to automatically produce a strategy such that the system can reach its objectives regardless of the scheduler behavior.

Acknowledgments

I would like to thanks B. Bérard, M. Potop-Butucaru, N. Sznajder, Y. Thierry-Mieg and S. Tixeuil for our helpful discussions, and R. Foulon for his previous work on robot protocols verification.

References

[1] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[2] L. Blin, A. Milani, M. Potop-Butucaru, and S. Tixeuil. Exclusive perpetual ring exploration without chirality. *Distributed Computing*, pages 312–327, 2010.

[3] S. Devismes. Optimal exploration of small rings. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security*, page 9, 2010.

[4] P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Principles of Distributed Systems*, pages 105–118, 2007.

[5] Roberto Grossi, Andrea Pietracaprina, and Geppino Pucci. Optimal deterministic protocols for mobile robots on a grid. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory SWAT'98*, volume 1432 of *Lecture Notes in Computer Science*, pages 181–192. Springer Berlin / Heidelberg, 1998.

[6] G. Prencipe. A new distributed model to control and coordinate a set of autonomous mobile robots: the CORDA model. Technical report, Universit di Pisa, 2000.

[7] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1996.

A Probabilistic Kleene Theorem

Benjamin Monmege
LSV, ENS Cachan & CNRS
Cachan, France
benjamin.monmege@lsv.ens-cachan.fr

Abstract

We provide a Kleene Theorem for (Rabin) probabilistic automata over finite words. Probabilistic automata generalize deterministic finite automata and assign to a word an acceptance probability. We provide probabilistic expressions with probabilistic choice, guarded choice, concatenation, and a star operator. We prove that expressions and automata are expressively equivalent. Our result actually extends to 2-way probabilistic automata with pebbles and corresponding expressions, as shown in [3].

1 Introduction

Kleene's Theorem states the equivalence of rational and recognizable languages in the free monoids. Naturally, this fundamental result has been generalized to various settings and in particular to quantitative extensions of classical languages, called formal power series [11, 10].

The present paper aims at a probabilistic counterpart of Kleene's Theorem. There are actually a variety of models for probabilistic systems, comprising Segala systems, generative systems, stratified systems, Markov chains, etc. (see [12] for an overview). Those models may involve non-determinism and *generate* some behavior according to probability distributions over states. Alternatively, they may make a probabilistic decision depending on the input letter, like (reactive) probabilistic automata the latter go back to Rabin [9] and are an object of ongoing research considering decision problems such as emptiness, language equivalence [11, 13], and the value 1 problem [7].

Our starting point of view is that expressions and automata shall represent quantitative properties of words. In particular, rather than at bisimulation equivalence, we are looking at language equivalence in terms of formal power series (i.e., mappings from strings to elements from the real-valued interval $[0, 1]$). This actually has an immediate impact on the choice of both the automaton model and the syntax of expressions that are supposed to characterize it. On the automata side, a probabilistic decision should depend on the *given* input. Therefore, we consider probabilistic automata. On the specification side, we would like to adopt concepts from rational expressions. We actually provide a simple fragment of classical weighted rational expressions over the nonnegative real numbers, including a star operator and concatenation. The star operator has to be handled with care, though. It comes with a subtle restriction to make sure that an expression associates with every word a probability. In this way, we obtain a class of probabilistic expressions that have the same expressive power as probabilistic automata. Translations forth and back are effective so that decidability results for automata directly carry over to expressions.

Actually, a more general result can be proved. Expressions can be extended in such a way that they capture 2-way probabilistic automata [5] and automata with pebbles (similar to 2-way word automata and tree-walking automata). Such expressions can then be considered as a probabilistic generalization of XPath. Note that (non-probabilistic) 2-way automata are in fact an appropriate machine model for

compiling XPath queries. The concept presented in this paper may therefore constitute a first step towards probabilistic database query languages: an expression is considered as a query, and an equivalent automaton can be used as a tool for evaluating queries efficiently (see [6] for recent developments on weighted query evaluation). This abstract is extracted from a joint work with Benedikt Bollig, Paul Gastin and Marc Zeitoun [3], in which these further developments have been studied in detail.

2 Probabilistic Automata and Expressions

We fix a finite alphabet A and consider words over A, i.e., sequences $w = a_0 \cdots a_{n-1} \in A^*$ with $n \geq 0$ and $a_i \in A$ for every i. The *length* n of w is denoted $|w|$.

Probabilistic Automata We consider classical probabilistic finite automata (PFA) [9] where we allow probabilities of acceptance. A PFA over alphabet A is a tuple $\mathscr{A} = (Q, \iota, Acc, \mathbb{P})$ where Q is the finite set of states, $\iota \in Q$ is the initial state, and $Acc\colon Q \to [0,1]$ is a function mapping every state to its probability of acceptance: in the following, a state q is said *accepting* if $Acc(q) > 0$. Moreover, $\mathbb{P} : Q \times A \times Q \to [0,1]$ is a function that assigns a probability to each transition. PFA are *reactive* automata, whose probabilistic choice depends on the current input letter. Thus, we require that $Acc(q) + \sum_{q' \in Q} \mathbb{P}(q,a,q') \leq 1$ for all $(q,a) \in Q \times A$. An *accepting run* of $\mathscr{A}$ over $w = a_0 \cdots a_{n-1} \in A^*$ is a sequence of transitions $\rho = \delta_1 \cdots \delta_n$ such that $\delta_i = (q_{i-1}, a_{i-1}, q_i)$ with $q_0 = \iota$ and q_n is an accepting state. For such a run ρ, we set $\mathbb{P}(\rho) = (\prod_{i=1}^{n} \mathbb{P}(\delta_i)) \times Acc(q_n)$. The semantics of the PFA $\mathscr{A}$ is the mapping (series) $[\![\mathscr{A}]\!] : A^+ \to [0,1]$ given by $[\![\mathscr{A}]\!](w) = \sum_\rho \mathbb{P}(\rho)$, where the sum ranges over all accepting runs ρ over w.

An example PFA is depicted on the right where 1 is the initial state and 3 is the only accepting state (with probability of acceptance being 1). Transitions with probability 0 are omitted. We have $[\![\mathscr{A}]\!](aab) = \frac{1}{4} + \frac{1}{6} = \frac{5}{12}$.

Probabilistic Expressions While PFAs are a machine model, we are aiming at denotational probabilistic regular expressions with the same expressiveness as PFAs. We start with the definition of classical weighted expressions (WEs):

$$E ::= s \mid a \mid E + E \mid E \cdot E \mid E^*$$

with $s \in \mathbb{R}_{\geq 0}$ and $a \in A$. We often simply write EF instead of $E \cdot F$. Also, we let $E^0 \stackrel{\text{def}}{=} 1$ and $E^{m+1} = EE^m$ for $m \geq 0$. The semantics of a WE E is a mapping $[\![E]\!] \colon A^* \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$ defined inductively by

$$[\![s]\!](w) = \begin{cases} s & \text{if } w = \varepsilon \\ 0 & \text{otherwise} \end{cases} \qquad [\![a]\!](w) = \begin{cases} 1 & \text{if } w = a \\ 0 & \text{otherwise} \end{cases} \qquad [\![E_1 \cdot E_2]\!](w) = \sum_{w=uv} [\![E_1]\!](u) \cdot [\![E_2]\!](v)$$

$$[\![E_1 + E_2]\!](w) = [\![E_1]\!](w) + [\![E_2]\!](w) \qquad [\![E^*]\!](w) = \sum_{m \in \mathbb{N}} [\![E^m]\!](w)$$

In the following, we consider expressions modulo the following trivial identities: $0 + E \equiv E + 0 \equiv E$, $E \cdot 0 \equiv 0 \cdot E \equiv 0$, $E \cdot 1 \equiv 1 \cdot E \equiv E$, $0^* \equiv 1$, as well as $s \cdot E \equiv E \cdot s$ (for $s \in \mathbb{R}_{\geq 0}$) which models commutativity.

We introduce below probabilistic regular expressions (PREs) as a fragment of WEs. We have to restrict WEs since otherwise values greater than 1 could be obtained. For instance, the WE $(a + ab)(ba + a)$ should not be a PRE since it evaluates to 2 on the word aba. The restriction will be both on sum and star. Since we aim at PREs which are equivalent to PFAs, let us examine first which type of WEs are obtained from PFAs. A transition $\delta = (q,a,q')$ with probability $\mathbb{P}(\delta) = s$ could be denoted by the expression sa. Applying classical algorithms to build a regular expression from finite-state automata, we then obtain, for the example automaton, the expression $\left[\frac{1}{6}a(a+b) + \frac{1}{2}a\right]^* \cdot (\frac{1}{3}a + b)$. Now, the expression $\left[\frac{1}{6}a(a+b) + \frac{1}{2}a\right]^* \cdot (a+b)$, obtained by changing the subexpression $\frac{1}{3}a$ into a, should be disallowed, because it corresponds to an automaton violating condition $Acc(q) + \sum_{q' \in Q} \mathbb{P}(q,a,q') \leq 1$. On the other

hand, $\left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (\frac{1}{3}a+\frac{1}{2}b)$ would be acceptable: we obtain a corresponding PFA from the previous automaton by setting $\mathbb{P}(1,b,3)=\frac{1}{2}$.

Definition 1. *Probabilistic regular expressions* (PREs) is the fragment of WEs defined as follows:

(Atoms) $p \in [0,1]$ and $a \in A$ are PREs.

$(+_a)$ If $(E_a)_{a\in A}$ are PREs, then $\sum_{a\in A} a \cdot E_a$ is a PRE (where we write the sum with left associativity).

$(+_p)$ If E and F are PREs and $p \in [0,1]$, then $p \cdot E + (1-p) \cdot F$ is a PRE.

$(\cdot)$ If E and F are PREs, then $E \cdot F$ is a PRE.

$(*)$ If $E+F$ is a PRE, then $E^* \cdot F$ is a PRE.

(ACD) Every WE that is obtained from a PRE by applying commutativity of $+$, associativity of $+$ or $\cdot$, or distributivity of $\cdot$ over $+$ is a PRE.

There are two *guarded* sums. The first one $(+_a)$ is deterministic and guarded by the next letter to be read. The second one $(+_p)$ is probabilistic. Also, the star operation contains an implicit choice which is either to iterate again the expression or to exit the loop. This choice also has to be guarded which is the reason for the precondition $E+F \in \mathsf{PRE}$ in the rule $(*)$. The guard could be deterministic as in $(ab)^*b$ or probabilistic as in $(\frac{1}{3}(aa+bb))^*\frac{2}{3}(a+b)$. Finally, with the above restrictions, we lose the classical ACD identities, hence we enforce these properties explicitely with the ACD-rules which allow to rewrite a PRE in order to apply the star rule as needed.

Since PREs form a fragment of WEs, the semantics is inherited. From Theorem 7 below, PREs are equivalent to PFAs. We deduce that the semantics of $E \in \mathsf{PRE}$ takes values in $[0,1]$, so that one can interpret $[\![E]\!](w)$ as a probability.

Example 2. A simple PRE is $(\frac{1}{3}a)^* \cdot \frac{2}{3}b$, which assigns to a word $a^m b$ the probability $(\frac{1}{3})^m \cdot \frac{2}{3}$, and 0 to words not in a^*b. Moreover, $E = \left[\frac{1}{6}a(a+b)+\frac{1}{2}a\right]^* \cdot (\frac{1}{3}a+b)$ is indeed a PRE for the automaton presented previously. To show that E is a PRE, we use some semantical equivalences such as $\frac{5}{6} \equiv \frac{1}{2}+\frac{1}{3}$. The expression $a(\frac{1}{6}(a+b)+\frac{5}{6})+b$ uses two deterministic sums (first and third) and a probabilistic sum. Using the above semantical equivalences and ACD-rules, we deduce that $\frac{1}{6}a(a+b)+\frac{1}{2}a+\frac{1}{3}a+b$ is a PRE and it remains to apply the star rule to get E.

3 A Probabilistic Kleene Theorem

In the following, we will show that, for a PFA, we can always find an equivalent PRE, and vice versa. This result, which is non-trivial even in the one-way setting, is generalized in [3] allowing two-way moves and pebbles. We first start by the translation from expressions to automata. The usual construction building standard automata, presented in [10] in the weighted case, cannot be directly used here. Indeed, notice that the rule $(*)$ is not an inductive rule: we need more information to apply the construction.

We start by defining the notion of terms of a PRE. Intuitively, if we suppose that summation is pushed up as much as possible by means of ACD-rules, then the multiset of terms consists of all expressions that occur in this big outermost sum. Formally, the definition is by induction over $E \in \mathsf{PRE}$. When E is an atom, we let $\mathsf{Terms}(E) = \{\{E\}\}$ be the singleton multiset containing only the atom itself. Moreover,[1]

$$\mathsf{Terms}(E+F) = \mathsf{Terms}(E) \uplus \mathsf{Terms}(F) \qquad \mathsf{Terms}(E^*) = \{\{E^*\}\}$$
$$\mathsf{Terms}(E \cdot F) = \{\{E' \cdot F' \mid E' \in \mathsf{Terms}(E), F' \in \mathsf{Terms}(F)\}\}$$

[1] Here, and in the following, we denote $A \uplus B$ the disjoint union of A and B.

Note that, if an expression F can be obtained from an expression E through ACD-rules, then we have $\mathsf{Terms}(E) = \mathsf{Terms}(F)$. The converse also holds as can be seen from the following proposition which can be easily proved by structural induction on the expression.

Lemma 3. *Let $E \in \mathsf{PRE}$ with $\mathsf{Terms}(E) = \{\{E_i \mid i \in I\}\}$. Using ACD-rules, we can rewrite E into $\sum_{i\in I} E_i$. In particular, we have $[\![E]\!] = \sum_{i\in I} [\![E_i]\!]$. Hence, we identify E and $\sum_{i\in I} E_i$.*

Proposition 4. *From any expression $E \in \mathsf{PRE}$ we can effectively construct an equivalent PFA $\mathscr{A} = (Q, \iota, Acc, \mathbb{P})$. More precisely, if $\mathsf{Terms}(E) = \{\{E_i \mid i \in I\}\}$, the set of accepting states of $\mathscr{A}$ is $\{f_i \mid i \in I\}$ and for all $i \in I$ the expression E_i is equivalent to the PFA $\mathscr{A}[f_i] = (Q, \iota, Acc', \mathbb{P})$, where $Acc'(f_i) = Acc(f_i)$ and $Acc'(q) = 0$ for all $q \neq f_i$.*

Proof. The construction is by structural induction on the expression $E \in \mathsf{PRE}$.

Automaton for PRE $p \in [0,1]$ is a single state with accepting probability p. For the PRE $a \in A$, the resulting automaton has two states, one initial and the other accepting with probability 1, and a single transition labeled by a in-between these two states.

Let $(E_a)_{a\in A}$ be PREs such that $\mathsf{Terms}(E_a) = \{\{E_{a,i} \mid i \in I_a\}\}$. By induction hypothesis, we have constructed suitable PFAs $\mathscr{A}_a = (Q_a, \iota_a, Acc_a, \mathbb{P}_a)$ with accepting states $\{f_{a,i} \mid i \in I_a\}$. Without loss of generality, we assume that the sets of states are pairwise disjoint. Consider $E' = \sum_{a\in A} a \cdot E_a$. We have $\mathsf{Terms}(E') = \biguplus_{a\in A} \{\{a \cdot E_{a,i} \mid i \in I_a\}\}$. We construct a PFA $\mathscr{A}' = (\biguplus_{a\in A} Q_a \uplus \{\iota'\}, \iota', \sum_{a\in A} Acc_a, \mathbb{P}')$ as the disjoint union of automata $\mathscr{A}_a$: moreover, from the new initial state ι', for every $a \in A$, we add an a transition with probability 1 to the state ι_a. We verify that $[\![\mathscr{A}']\!] = [\![E']\!]$.

Now let $E, E' \in \mathsf{PRE}$ be such that $\mathsf{Terms}(E) = \{\{E_i \mid i \in I\}\}$ and $\mathsf{Terms}(E') = \{\{E'_j \mid j \in J\}\}$. By induction hypothesis, we have constructed two suitable PFAs $\mathscr{A} = (Q, \iota, Acc, \mathbb{P})$ and $\mathscr{A}' = (Q', \iota', Acc', \mathbb{P}')$ with respective accepting states $\{f_i \mid i \in I\}$ and $\{f'_j \mid j \in J\}$. We assume that $Q \cap Q' = \emptyset$.

The terms of $E'' = p \cdot E + (1-p) \cdot E'$ are $\mathsf{Terms}(E'') = \{\{p \cdot E_i \mid i \in I\}\} \uplus \{\{(1-p) \cdot E'_j \mid j \in J\}\}$. Let $\mathscr{A}'' = (Q \uplus Q' \uplus \{\iota''\}, \iota'', Acc'', \mathbb{P}'')$ be the PFA defined as the disjoint union of $\mathscr{A}$ and $\mathscr{A}'$, with additional transitions exiting state ι'' defined by $\mathbb{P}''(\iota'', a, q) = p\mathbb{P}(\iota, a, q)$ if $q \in Q$ and $\mathbb{P}''(\iota'', a, q) = (1-p)\mathbb{P}'(\iota', a, q)$ if $q \in Q'$; moreover, we set $Acc''(q) = Acc(q)$ if $q \in Q$, $Acc''(q) = Acc'(q)$ if $q \in Q'$ and $Acc''(\iota'') = Acc(\iota) \times Acc'(\iota')$. We can verify that $\mathscr{A}''$ is indeed a PFA such that $[\![\mathscr{A}'']\!] = [\![E'']\!]$.

For the concatenation, let $E'' = E \cdot E'$, which has as terms $\mathsf{Terms}(E'') = \{\{E_i \cdot E'_j \mid (i,j) \in I \times J\}\}$. The automaton $\mathscr{A}''$ for E'' consists of one copy of $\mathscr{A}$ and a copy $\mathscr{A}'_i$ of $\mathscr{A}'$ for every $i \in I$. First, $\mathscr{A}''$ simulates $\mathscr{A}$ until it reaches some accepting state f_i of $\mathscr{A}$. Then, we duplicate each transition from ι' to a state q of $\mathscr{A}'$, to a transition from f_i to the copy of q in $\mathscr{A}_i$, with probability the product of the previous probability and $Acc(f_i)$. Once again, $\mathscr{A}''$ can be shown to be a PFA equivalent to E''.

For the Kleene star we assume that $E = F + G$ and $E'' = F^* \cdot G$. We have $I = K \uplus L$ with $\mathsf{Terms}(F) = \{\{E_i \mid i \in K\}\}$ and $\mathsf{Terms}(G) = \{\{E_i \mid i \in L\}\}$. Hence, we have $\mathsf{Terms}(E'') = \{\{F^* \cdot E_i \mid i \in L\}\}$. We construct the PFA $\mathscr{A}'' = (Q, \iota, Acc'', \mathbb{P}'')$ with accepting states $\{f_i \mid i \in L\}$ by duplicating each transition reading a from ι to a state q into a transition reading a from state f_i ($i \in K$) to state q, with probability being the product of $\mathbb{P}(\iota, a, q)$ and $Acc(f_i)$.

Finally, if E'' is obtained from E via ACD-rules, we have $\mathsf{Terms}(E'') = \mathsf{Terms}(E)$ so we can keep the same automaton: $\mathscr{A}'' = \mathscr{A}$. □

In order to construct a PRE which is equivalent to a PFA, we need to be able to concatenate a PRE after an arbitrary term of another PRE. This is possible thanks to the following result.

Lemma 5. *If $E + F$ and G are PREs, then $E + F \cdot G$ is also a PRE.*

Proposition 6. *For every PFA $\mathscr{A}$ there is an equivalent PRE E.*

Proof. Let $\mathscr{A} = (Q, \iota, Acc, \mathbb{P})$ be a PFA. We construct a PRE $E_q = Acc(q) + \sum_{q' \in Q} E_{q,q'} Acc(q')$ for each $q \in Q$, where $[\![E_{q,q'}]\!](w)$ computes the sum of the probabilities of *nonempty* runs over w starting from state q, ending in state q' (without taking into account the acceptance probability of q'). Hence, we will obtain the PRE E_ι, which computes exactly the behavior of $\mathscr{A}$.

We follow usual procedures to translate automata into expressions. For $q' \in Q$ and $X \subseteq Q$, we define $Acc^X_{q'} = Acc(q')$ if $q' \in X$ and 1 otherwise. For $q \in Q$ and $X \subseteq Q$, we construct by induction on X a PRE $E^X_q = Acc(q) + \sum_{q' \in Q} E^X_{q,q'} Acc^X_{q'}$ where $E^X_{q,q'}$ is a PRE such that $[\![E^X_{q,q'}]\!](w)$ is the sum of the probabilities of *nonempty* runs over w starting from state q, ending in state q' and using only *intermediary* states in X. Hence, we have $E_q = E^Q_q$ and $E_{q,q'} = E^Q_{q,q'}$.

The base of the induction is when $X = \emptyset$. For each state $q \in Q$ and letter $a \in A$, by definition of PFAs we have $Acc(q) + \sum_{q' \in Q} \mathbb{P}(q,a,q') \leq 1$. Hence, using rules $(+_a)$ and $(+_s)$ we obtain the PRE

$$E^\emptyset_q = Acc(q) + \sum\nolimits_{a \in A} a \cdot \sum\nolimits_{q' \in Q} \mathbb{P}(q,a,q') = Acc(q) + \sum\nolimits_{q' \in Q} E^\emptyset_{q,q'} Acc^\emptyset_{q'}$$

where the last equality is obtained using ACD-rules and $Acc^\emptyset_{q'} = 1$.

For the induction step, let $r \in Q \setminus X$. By induction, we assume that PREs E^X_q have been constructed for all $q \in Q$, and we construct $E^{X \cup \{r\}}_q$. We have $E^X_r = Acc(r) + \sum_{q' \in Q} E^X_{r,q'} Acc^X_{q'} \in \mathsf{PRE}$ and $Acc^X_r = 1$ since $r \notin X$. Using rule $(*)$, we get $G^X_r = \left(E^X_{r,r}\right)^* \cdot \left(Acc(r) + \sum_{q' \in Q \setminus \{r\}} E^X_{r,q'} Acc^X_{q'}\right) \in \mathsf{PRE}$. Now, $E^X_q = Acc(q) + \sum_{q' \in Q} E^X_{q,q'} Acc^X_{q'} \in \mathsf{PRE}$ and $Acc^X_r = 1$. Using Lemma 5, we can plug G^X_r after $E^X_{q,r}$. Using ACD-rules, we obtain the PRE

$$\begin{aligned} E^{X \cup \{r\}}_q &= Acc(q) + E^X_{q,r} \cdot G^X_r + \sum\nolimits_{q' \in Q \setminus \{r\}} E^X_{q,q'} Acc^X_{q'} \\ &= Acc(q) + E^X_{q,r} (E^X_{r,r})^* Acc(r) + \sum\nolimits_{q' \in Q \setminus \{r\}} \left(E^X_{q,q'} + E^X_{q,r} \left(E^X_{r,r}\right)^* E^X_{r,q'}\right) Acc^X_{q'} \\ &= Acc(q) + \sum\nolimits_{q' \in Q} E^{X \cup \{r\}}_{q,q'} Acc^{X \cup \{r\}}_{q'} \end{aligned}$$

using $Acc^{X \cup \{r\}}_r = Acc(r)$ and $Acc^{X \cup \{r\}}_{q'} = Acc^X_{q'}$ if $q' \in Q \setminus \{r\}$. □

We have finally proved that

Theorem 7. PFAs *and* PREs *are effectively equivalent.*

With Theorem 7, decidability of the equivalence problem for PFAs carries over to PREs (provided the probabilities in an expression are rational numbers), whereas their threshold problem is undecidable (as this problem is undecidable for automata, as originally proved in [8]). We could also state further undecidability results about isolated cutpoints using, e.g., results of [1].

Corollary 8. *The equivalence problem for* PREs *is decidable: given* PREs E *and* F*, does* $[\![E]\!] = [\![F]\!]$ *hold? The threshold problem for* PREs *is undecidable: given an alphabet* A*, a* PRE E *over* A *and* $0 < s < 1$*, is there a word* $w \in A^+$ *such that* $[\![E]\!](w) \geq s$*?*

Note that PFAs cannot recognize all series recognized by usual Rabin automata, i.e., PFAs without the probabilities over accepting states. For example, the map $g \colon A^* \to [0,1]$, defined by $g(a^n) = 1$ if $n > 0$ and $g(w) = 0$ for all other words w, is not recognizable by a PFA (note that a^*a is not a PRE). However, g is recognized by a Rabin automaton with a single state. To deal with this issue, we can add a fresh symbol $\lhd$ at the end of a word. For a function $f \colon A^* \to [0,1]$, we define $f_\lhd \colon (A \cup \{\lhd\})^* \to [0,1]$ by $f_\lhd(w\lhd) = f(w)$ if $w \in A^*$, and 0 otherwise. For example, the series $g_\lhd$ is definable by $a(a^*\lhd)$, which is a PRE since $a + \lhd \in \mathsf{PRE}$. More generally, we can prove the following:

Proposition 9. *Let* $f \colon A^* \to [0,1]$*. The function* $f_\lhd$ *is recognizable by a* PFA *(or equivalently by a* PRE*) iff* f *is recognizable by a Rabin automaton.*

4 Conclusion

In this abstract, we presented a probabilistic Kleene Theorem for classical probabilistic automata, that can be extended for automata with two-way navigation and pebbles, as we did in [3]. As applications, this is particularly useful to translate probabilistic linear temporal logic (which computes probability for sets of paths defined by usual LTL operators) into automata. This constitutes a first step towards probabilistic XPath, so we aim at extending our work to finite trees and probabilistic tree automata. Notice that construction of Proposition 4 leads to an exponentially large automaton: however, we believe this construction can be improved to get a linear-size equivalent automaton. Just like classical finite automata, weighted automata over semirings enjoy characterizations in terms of monadic second-order logic [4, 2]. Continuing this line of research, a recent paper establishes a logical characterization of probabilistic automata [14]. It would be interesting to study whether alternative characterizations exist that use, for example, a transitive-closure operator [2].

References

[1] Alberto Bertoni, Giancarlo Mauri, and Mauro Torelli. Some recursive unsolvable problems relating to isolated cutpoints in probabilistic automata. In *Proc. of ICALP'77*, pages 87–94, 1977.

[2] B. Bollig, P. Gastin, B. Monmege, and M. Zeitoun. Pebble weighted automata and transitive closure logics. In *Proc. of ICALP'10*, volume 6199 of *LNCS*, pages 587–598. Springer, 2010.

[3] B. Bollig, P. Gastin, B. Monmege, and M. Zeitoun. A probabilistic Kleene theorem. In *Proc. of ATVA'12*, LNCS, pages 400–415. Springer, 2012.

[4] M. Droste and P. Gastin. Weighted automata and weighted logics. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, chapter 5, pages 175–211. Springer, 2009.

[5] C. Dwork and L. Stockmeyer. On the power of 2-way probabilistic finite state automata. In *Proc. of FoCS'89*, pages 480–485. IEEE Computer Society, 1989.

[6] P. Gastin and B. Monmege. Adding pebbles to weighted automata. In *Proc. of CIAA'12*, LNCS. Springer, 2012.

[7] H. Gimbert and Y. Oualhadj. Probabilistic automata on finite words: Decidable and undecidable problems. In *Proc. of ICALP'10*, volume 6199 of *LNCS*, pages 527–538. Springer, 2010.

[8] A. Paz. *Introduction to probabilistic automata*. Academic Press, 1971.

[9] M. O. Rabin. Probabilistic automata. *Inform. and Control*, 6:230–245, 1963.

[10] J. Sakarovitch. Rational and recognizable power series. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, chapter 4, pages 103–172. Springer, 2009.

[11] M. P. Schützenberger. On the definition of a family of automata. *Inform. and Control*, 4:245–270, 1961.

[12] R. Segala. Probability and nondeterminism in operational models of concurrency. In *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 64–78. Springer, 2006.

[13] W.-G. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM J. Comput.*, 21(2):216–227, 1992.

[14] Th. Weidner. Probabilistic automata and probabilistic logic. In *Proc. of MFCS'12*, LNCS. Springer, 2012.

Constraint-based Static Analyses for Java Bytecode Programs

Đurica Nikolić
Dipartimento di Informatica, University of Verona
The Microsoft Research - University of Trento Center for Computational and Systems Biology
durica.nikolic@univr.it

Abstract

In this paper we illustrate the general idea of a parameterized formal framework for interprocedural static analyses of Java bytecode programs, based on abstract interpretation. The latter allows us to show, under hypotheses we characterize, the soundness of the analyses belonging to the framework. Our framework is capable of dealing with side-effects and exceptional executions of invoked methods. Different instantiation of the parameters of our formal framework give rise to different static analyses. It is necessary to show, for each instantiation, that it satisfies the hypotheses mentioned above. This way, the general theory of the framework guarantees the soundness of the new analysis obtained from the instantiation. We introduce two static analyses formalized this way: Reachability Analysis of Program Variables and Definite Expression Aliasing Analysis, which have been also implemented in the static analyzer Julia and which improved the precision of its principal tools.

1 Introduction

Static analysis of computer programs allows one to gather information about the runtime behavior of such programs, before they are run. One of its goals is to prove that programs do not perform any illegal operation, such as a division by zero or a dereference of `null`, or do not lead to erroneous executions, such as infinite loops, or do not divulge information in incorrect ways (such as security authorizations or GPS position in mobile devices). Static analysis has a long story now and can be formalized in many ways. We follow the abstract interpretation [1] approach, which allows one to define a static analysis from the formal specification of the property of interest and of the semantics of the programming language.

Dynamic allocation of objects is heavily used in (complex and large) real life programs. When such objects are instantiated on demand, their number might not be statically known. Moreover, objects in general contain references to other objects (i.e., *fields* in object-oriented parlance) and those references are usually updated at run-time. The most interesting properties of the present software products are related to the objects they dynamically allocate in memory rather than to primitive values such as integers. A large literature tackles the analysis of memory-related properties. There are very general techniques, such as *shape analysis*, that statically build a conservative description of possible *shapes* that data structures might have at runtime. There exist some more abstract analyses, typically less precise but more efficient. For instance, *aliasing analysis* exists in uncountable variations and expresses the fact that two variables might (or must always) point to the same location (i.e., they are possible or definite *alias* of each other), while *sharing analysis* determines whether two variables might ever be bound to overlapping data structures, i.e., two variables share if they might *reach* the same location at runtime.

We introduce a formal framework for static analyses of Java bytecode programs capable of capturing the memory-related properties mentioned above. Our analyses are constraint-based: a large constraint is built from the program, whose solution is a sound approximation of the property of interest at each program point. The correctness of our analyses is proved in the abstract interpretation framework. We also

show two non-trivial static analyses obtained as instantiations of our framework: *Reachability Analysis of Program Variables* [6] and *Definite Expression Aliasing Analysis* [5].

2 Formal Framework for Constraint-based Analysis

In this section we introduce a general idea of our approach. First of all we introduce our target, Java bytecode-like language, used in [7, 6, 5] and inspired by the standard informal semantics [2]. It contains the following instructions: const x, dup, load, store, inc, add, sub, mul, div, rem, ifeq, ifne, new, getfield, putfield, throw and call. They abstract whole classes of Java bytecode instructions such as iconst_x, ldc, bipush, dup, iload, aload, istore, astore, iinc, iadd, isub, imul, idiv, irem, ifeq, ifne, if_null, if_nonnull, new, getfield, putfield, athrow, invokevirtual and invokespecial. Moreover, the instruction catch starts the exception handlers. We analyze programs at bytecode level for several reasons: there is a small number of bytecode instructions, compared to varieties of source statements; bytecode lacks complexities such as inner classes; implementations of our analyses are at bytecode level as well, which brings formalism, implementation and correctness proofs closer. We analyze bytecode preprocessed into a control flow graph (CFG), i.e., a directed graph of *basic blocks*, with no jumps inside them. Operational semantics of our target language corresponds to the standard operational semantics of Java bytecode [2].

Example 1. In Fig. 1 we show a simple Java method `delayMinBy` of a class `Event` (at the left) and its corresponding CFG (at the right). ■

Abstract Interpretation. Our framework is based on the general theory of approximations, called abstract interpretation [1]. Its general idea is the following. Suppose we are given a set of elements, called *concrete domain* and a function f operating on the latter. Abstract interpretation defines an *abstract domain*, i.e., the set of elements containing less pieces of information comparing to the concrete ones, two maps relating the domains and and a function operating on the abstract domain which simulates f. In our framework, all the analyses use the same concrete domain C which is composed of all possible states that can be verified in a program point. On the other hand, the abstract domain A depends on the property we are interested in, and its elements are obtained by extracting away from the concrete states all those pieces of information which are irrelevant for the analysis of interest. For every analysis, it is necessary to formally define the form of abstract elements and an ordering among them. The former determines the type of information our analysis captures, while the latter is relevant for the existences and uniqueness of the solution of our analysis. Moreover, concrete and abstract elements have to be connected by a pair of functions $\alpha : \mathsf{C} \to \mathsf{A}$ (*abstraction*) and $\gamma : \mathsf{A} \to \mathsf{C}$ (*concretization*) which specify which concrete elements correspond to which abstract elements (γ) and vice versa (α).

Abstract Constraint Graph. Our analysis is constraint-based: we construct an *abstract constraint graph* from the program under analysis and then we solve these constraints. For each bytecode instruction of the program under analysis there is a node containing an approximation of the information related to the property of interest at that point. Arcs of the graph propagate these approximations, reflecting, in abstract terms, the effects of the concrete semantics on the property of interest. In other words, an arc

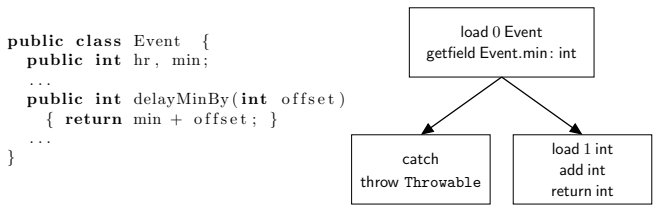


Figure 1: Our running example

between the nodes corresponding to two instructions b_1 and b_2 propagates the approximation at b_1 into that at b_2. The exact meaning of *propagates* depends on: (i) the property we are interested in and the abstraction we use to represent it; (ii) the type of analysis we want to perform: possible or definite; (iii) b_1 itself, since different instructions have different effects on the approximations present at that point.

Definition 2 (ACG)**.** Let P be the program under analysis in the form of a CFG for each method or constructor. The *abstract constraint graph* (ACG) for P is a directed graph $\langle V, E\rangle$ (nodes, arcs) where: (i) V contains a node $\boxed{\mathsf{ins}}$ for each instruction ins in P; (ii) for each method or constructor m in P, V contains nodes $\boxed{\mathsf{exit@}m}$ and $\boxed{\mathsf{exception@}m}$, representing the normal and the exceptional ends of m; (iii) each node contains an abstract element representing an approximation of the information related to the property of interest at that point; (iv) E contains directed arcs with one (1−1) or two (2−1) sources and always one sink. Each arc has a *propagation rule*, i.e., a function over the abstract domain, from the approximation(s) of its source(s) to the one of its sink. We distinguish the following types of arcs:

- **Sequential arcs:** if ins is an instruction in P, distinct from call, immediately followed by an instruction ins′, distinct from catch, then an 1−1 sequential arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{ins'}}$;
- **Final arcs:** for each return t and throw κ instructions occurring in a method or a constructor m of P, there are 1−1 final arcs from $\boxed{\mathsf{return\ t}}$ to $\boxed{\mathsf{exit@}m}$ and from $\boxed{\mathsf{throw}\,\kappa}$ to $\boxed{\mathsf{exception@}m}$, respectively;
- **Exceptional arcs:** for each instruction ins throwing an exception, immediately followed by a catch, an 1−1 exceptional arc is built from $\boxed{\mathsf{ins}}$ to $\boxed{\mathsf{catch}}$;
- **Parameter passing arcs:** for each call $m_1 \dots m_q$ occurring in P with π parameters (including the implicit parameter `this`) we build, for each $1 \le w \le q$, an 1−1 parameter passing arc from $\boxed{\mathsf{call}\ m_1 \dots m_q}$ to the node corresponding to the first bytecode instruction of the method m_w;
- **Return value arcs:** for each call ins_C = call $m_1 \dots m_q$ to a method with π parameters (including parameter `this`) returning a value of type t ≠ void, and each subsequent instruction ins_N distinct from catch, we build, for each $1 \le w \le q$, a 2 − 1 return value arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$ to $\boxed{\mathsf{ins}_N}$;
- **Side-effects arcs:** for each call ins_C = call $m_1 \dots m_q$ to a method with π parameters (including the implicit parameter `this`), and each subsequent bytecode instruction ins_N, we build, for each $1 \le w \le q$, a 2 − 1 side-effects arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$ to $\boxed{\mathsf{ins}_N}$, if ins_N is not a catch and a 2 − 1 side-effect arc from $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exception@}m_w}$ to $\boxed{\mathsf{catch}}$.

The sequential arcs correspond to the non-exceptional executions of all the instructions except call, return and throw. The final arcs connect the nodes corresponding to the last instruction of each method or constructor m (i.e., return or throw) to the special nodes $\boxed{\mathsf{exit@}m}$, in the case of return, and $\boxed{\mathsf{exception@}m}$, in the case of throw. The exceptional arcs represent the exceptional executions of the instructions that might launch an exception, i.e., call, new, throw, getfield and putfield, and they connect the nodes corresponding to these instructions with the node related to the catch instruction at the beginning of their exceptional handlers. The parameter passing arcs link every node corresponding to a method call to the node corresponding to the first instruction of the method(s) that might be called there. There exists a return value arc for each dynamic target m of a call ins_C returning a value. These arcs have two sources, $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m_w}$, and they propagate the approximations present at these nodes to the node corresponding to the bytecode instruction following ins_C. Moreover, these arcs might enrich the resulting approximation with some additional abstract elements due to the m's returned value. The execution of the method m might modify the memory in which m is executed and this might affect the approximation at the node $\boxed{\mathsf{ins}_C}$ corresponding to the method call ins_C. The side-effects arcs deal with these phenomena, i.e., they are 2−1 arcs connecting $\boxed{\mathsf{ins}_C}$ and $\boxed{\mathsf{exit@}m}$ (respectively $\boxed{\mathsf{exception@}m}$) with the

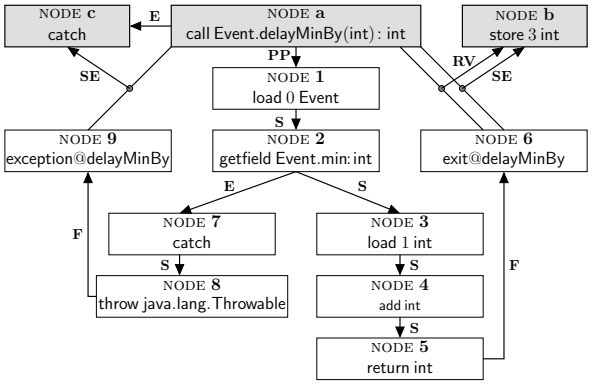


Figure 2: The ACG for the method `delayMinBy` in Fig. 1

$$
\begin{array}{rcl}
\Pi^{PP}(A_{\mathbf{a}}) & \sqsubseteq & A_{\mathbf{1}} \\
\Pi^{S}(A_{\mathbf{1}}) & \sqsubseteq & A_{\mathbf{2}} \\
\Pi^{S}(A_{\mathbf{2}}) & \sqsubseteq & A_{\mathbf{3}} \\
\Pi^{S}(A_{\mathbf{3}}) & \sqsubseteq & A_{\mathbf{4}} \\
\Pi^{S}(A_{\mathbf{4}}) & \sqsubseteq & A_{\mathbf{5}} \\
\Pi^{F}(A_{\mathbf{5}}) & \sqsubseteq & A_{\mathbf{6}} \\
\Pi^{E}(A_{\mathbf{2}}) & \sqsubseteq & A_{\mathbf{7}} \\
\Pi^{S}(A_{\mathbf{7}}) & \sqsubseteq & A_{\mathbf{8}} \\
\Pi^{F}(A_{\mathbf{8}}) & \sqsubseteq & A_{\mathbf{9}} \\
\Pi^{RV}(A_{\mathbf{a}}, A_{\mathbf{6}}) \sqcup \Pi^{SE}(A_{\mathbf{a}}, A_{\mathbf{6}}) & \sqsubseteq & A_{\mathbf{b}} \\
\Pi^{E}(A_{\mathbf{6}}) \sqcup \Pi^{SE}(A_{\mathbf{a}}, A_{\mathbf{6}}) & \sqsubseteq & A_{\mathbf{c}}
\end{array}
$$

Figure 3: The constraints extracted from the ACG of Fig. 2

node corresponding to the instruction (respectively catch) which follows ins_C, for each dynamic target m of the call, and propagate the approximation at $\boxed{\mathsf{ins}_C}$ modified by the side-effects of m's execution.

Example 3. In Fig. 2 we give the ACG of the method `delayMinBy` from Fig. 1. Nodes **a**, **b** and **c** belong to the caller of this method and exemplify the arcs related to the call and return bytecodes. Arcs are decorated with an abbreviation denoting their types: **S**, **F**, **E**, **PP**, **RV** and **SE** state for sequential, final, exceptional, parameter passing, return value and side-effects arcs respectively. ■

Each arcs of the ACG is associated with a function showing how the approximations at its sources are transformed in the approximation of its sink. We call these functions *propagation rules*. Their formal definition depends on a concrete property we are interested in and it represents the actual static analysis we perform. Since our goal is to introduce a general framework for constraint-based static analyses of Java bytecode programs, we do not concentrate on one particular property of interest, and therefore we do not provide the formal definition of the propagation rules related to that property.

Constraints. The ACG of the program under analysis introduces, for each of its nodes a set of constraints: one for every in-going arc. Every correct solution of these constraints is one possible result of our static analysis. The following definition shows how the constraints are extracted from an ACG.

Definition 4 (Constraints)**.** Let N be a node of an ACG and A_N the approximation of the property of interest information contained in that node. Suppose that there are k arcs whose sink is N and for each $1 \leq i \leq k$, let Π^i and $\mathsf{approx}(i)$ respectively denote the propagation rule and the approximation of the property of interest at the source(s) of the i^{th} arc. These arcs give rise to the following constraint: $\sqcup_{i=1}^{k} \Pi^i(\mathsf{approx}(i)) \sqsubseteq A_N$, where $\sqsubseteq$ and $\sqcup$ are the partial ordering and join operator of the abstract domain.

Example 5. In Fig. 3, we show the constraints extracted from the ACG introduced in Example 3. These constraints concern the method `delayMinBy` only, and not the whole program under analysis. ■

We showed that, when A satisfies the ACC condition, and when the propagation rules are monotonic, then there exists a solution of the constraints constructed this way and it is unique.

Soundness. We formally proved that, when the propagation rules correctly simulate the bytecode instructions corresponding to their sources, our static analyses are sound. Due to space limitations we do not provide the theorem stating that result, since it requires a lot of formal definitions not included in this paper. The proof uses the most important results of the abstract interpretation framework.

3 Examples of Constraint-based Analyses

In this section we introduce two static analysis of Java bytecode programs which can be instantiated in the framework of constraint-based analyses.

3.1 Reachability Analysis of Program Variables

In [6], we presented a new abstract domain for the static analysis of reachability between program variables, through dynamically allocated memory locations. Reachability from a program variable v to a program variable w in a state σ states that from v it is possible to follow a path of memory locations (specified by σ) that leads to the object bound to w. This information is important for improving the precision of other static analyses, such as side-effects, field initialization, cyclicity and path-length analysis, as well as more complex analyses built upon them, such as nullness and termination analysis of our static analyzer Julia (`www.juliasoft.com`). Our *Reachability Analysis* is a static analysis which determines, for each program point p of a Java bytecode program, a set of pairs of variables $\langle v, w \rangle$ such that v might reach w at p, i.e., an over-approximation of the actual reachability information available at p. Therefore, this is a *possible* analysis of our framework. The approximation obtained this way is state-independent, i.e., it holds for any possible execution of the program. We formally prove the soundness of this analysis by showing that all the requirements specified by the framework introduced in the previous section (e.g., ACC condition on the abstract domain, monotonicity and consistence of propagation rules) are satisfied. The formal proof of correctness can be found in the extended version of the paper, which has been submitted for publication [4]. We have implemented the analysis inside Julia. Our experiments of analysis of non-trivial Java and Android programs (written by Google) show the improvement of precision due to the presence of reachability information. Figure 4 presents our experiments with the nullness tool of Julia when our reachability analysis is included and excluded. In 8 cases over 24, the reachability information improves the precision of the nullness tool. Moreover, the presence of the reachability information actually reduces the total runtime of the tool. This is because reachability helps subsequent analyses, in particular side-effects analysis, and prevents them from generating too much spurious information.

3.2 Definite Expression Aliasing Analysis

Another novel static analysis for Java bytecode, called *Definite Expression Aliasing* has been introduced in [5]. It infers, for each variable v at each program point p, a set of expressions whose value at p is always equal to the value of v at p, for every possible execution of the program. Namely, it determines which expressions *must* be aliased to the local variables and the stack elements of the Java Virtual Machine available at the point of interest. This is a useful piece of information for an inter-procedural static analyzer, such as Julia, since it can be used to refine other analyses at conditional statements or assignments. This *definite* constraint-based static analysis has been formalized (by instantiating different parameters of our formal framework) and implemented inside the Julia tool. Moreover, in the extended version of the paper [3], submitted for publication, we show that our Definite Expression Aliasing Analysis is correct by showing that different requirements specified by our framework are satisfied. We have shown the benefits of our analysis for nullness and termination analyses with Julia by analysing some real-life benchmarks. For example, we use our analysis at the then branch of each comparis-

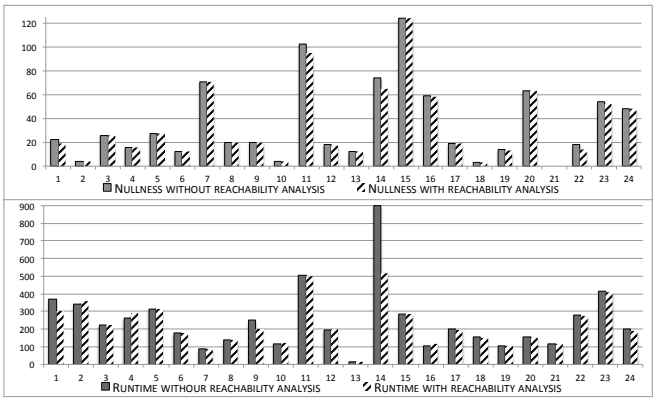


Figure 4: Comparison of the number of warnings (possible dereference of `null`, possibly passing `null` to a library method) produced by the nullness tool of Julia (top) and of the runtimes (in seconds) of that tool (bottom) when our reachability analysis is present and absent

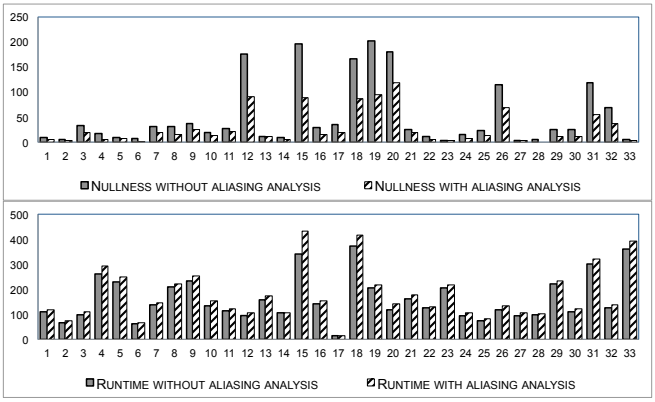


Figure 5: Comparison of the number of warnings (possible dereference of `null`, possibly passing `null` to a library method) produced by the nullness tool of Julia (top) and of the runtimes (in seconds) of that tool (bottom) when our definite aliasing analysis is present and absent

son `if (v!=null)` to infer that the definite aliases of `v` are non-`null` there, and at each assignment `w.f=exp` to infer that expressions `E.f` are non-`null` when `exp` is non-`null` and when `E` is a definite alias of `w` whose evaluation does not read nor write `f`. Figure 5 reports the precision and the runtime of our nullness analysis: we performed it first without and then with the help of our aliasing analysis. A clear difference between the two runs is that the runtime of the nullness analysis increased by 9.88%, when the definite expression aliasing analysis is activated, but its precision improved by 45.98%.

4 Conclusion

Our parametrized formal framework for non-trivial, interprocedural constraint-based static analyses of Java bytecode programs has found various instantiation inside of our static analyzer Julia. Field Initialization, Reachability of Program Variables, Sharing, Definite Aliased Expressions, Side-Effects are only some of Julia's static analyses obtained as particular instantiation of our framework.

References

[1] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages - POPL*, pages 238–252. ACM, 1977.

[2] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[3] Đ. Nikolić and F. Spoto. Definite Expression Aliasing in Java Bytecode Programs: a Constraint-Based Static Analysis - submitted. `http://profs.sci.univr.it/~nikolic/download/ICTAC2012/ICTAC2012Ext.pdf`.

[4] Đ. Nikolić and F. Spoto. Reachability Analysis of Program Variables - submitted. `http://profs.sci.univr.it/~nikolic/download/IJCAR2012/IJCAR2012Ext.pdf`.

[5] Đ. Nikolić and F. Spoto. Definite Expression Aliasing Analysis for Java Bytecode. In *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing - ICTAC*, volume 7521 of *LNCS*, pages 74–89. Springer, 2012.

[6] Đ. Nikolić and F. Spoto. Reachability Analysis of Program Variables. In *Proceedings of the 6th International Joint Conference on Automated Reasoning - IJCAR*, volume 7364 of *LNAI*, pages 423–438. Springer, 2012.

[7] F. Spoto and M. D. Ernst. Inference of Field Initialization. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 231–240. ACM, 2011.

Tree-Regular Analysis of Parallel Programs with Dynamic Thread Creation and Locks

Benedikt Nordhoff
Westfälische Wilhelms-Universität*
Münster, Germany
b.n@wwu.de

Abstract

We utilize Monitor-DPN to precisely model concurrent programs with unbounded recursion, dynamic thread creation and synchronization via well-nested locks based on finite abstractions of procedure- and thread-local state. One obtains a tree-regular characterization of reachable configurations by using acquisition structures to check for lock-sensitive schedulability. This technique can then be iterated using additional release structures. This then allows to solve for instance bit-vector problems. The techniques have been implemented for the analysis of Java programs and applied for data race detection and improvement of an information flow control analysis.

1 Introduction

Esparza and Knoop [3] demonstrated how automata theoretic approaches can be used to solve data-flow problems like bit-vektor analyses. We utilize an automata theoretic approach which handles full recursion, thread creation and synchronization via reentrant well-nested locks. The underlying model is that of Dynamic Pushdown Networks (DPN) which where introduced (without locks) by Bouajjani et al. [1] in 2005. They showed how the pre^* operator for calculating predecessor sets of configurations – words over an alphabet of control and stack symbols – preserves regularity. Later the DPN-model was enriched by Lammich et al. [8] to allow for scheduling restrictions based on well-nested locks. This was done by extending techniques from Kahlon et al. [7, 6] and thereby obtaining a tree-regular characterization of lock-sensitive schedulability. This could then be encoded into the control states of the DPN. The converse set of configurations reachable from a given configuration ($post^*$) was shown to be non-regular in the word semantics [1]. Gawlitza et al. [4] then switched from configuration words to *execution trees*. An execution tree describes the steps of an execution as well as the reached configuration. The tree structure makes visible both the parallel execution of steps in different threads and the nesting structure of procedure calls and returns. This allowed them to obtain a tree-regular representation of the set of execution trees reachable from a fixed initial single thread configuration. Here we describe recent extensions [9] which also allow to obtain such a representation for arbitrary tree-regular sets of reachable configurations, this again allows to solve for instance bit-vector problems.

Motivation

This work was motivated by an application for an information flow control analysis for parallel Java programs based on system dependence graphs (SDG) as described for example by Giffhorn [5]. Besides

*This work was founded by the DFG in project IFC for Mobile Components within priority program RS^3 (SPP 1496).

#	main method	t2's run method
1	`print(x);` `t2.start();`	`x = 42;`

2	`synchronized(a){`
	`  t2.start();`
	`  print(x);  }`
		`synchronized(a){}`
		`x = 42;`

3	`t2.start();`
	`synchronized(a){`
	`  x = 17;`
	`  print(x);  }`
		`synchronized(a){`
		`  x = 42;`
		`}`

#	main method	t2's run method

4	`t2.start();`
	`synchronized(a){`
	`  y = 42;`
	`  print(x);  }`
		`synchronized(a){`
		`  x = y;`
		`}`

5	`t2.start()`
	`synchronized(a){`
	`  synchronized(b){`
	`  x = 42;  }`
	`  x = 17;  }`
		`synchronized(b){`
		`  if (...) {`
		`  synchronized(a){}`
		`  } else  {`
		`  x = 17; }`
		`  print(x);  }`

Table 1: Spurious examples of data flows between threads

obtaining the first implementation for the previously developed techniques for real world applications, the goal was to improve the handling of data flows between different threads in the SDG based analysis. To illustrate the effects of locking and thread creation, consider the snippets of Java code in Table 1. Assume the left code resides in the `main`-method and the right code in the `run`-method of a `Thread` object `t2`. That is execution starts with the code on the left in which `t2.start()` is called to start parallel execution of the code on the right. The question for the analysis is: Can the programs print the value 42? An analysis abstracting from thread creation would spuriously assume so in the first example. The second example shows the interplay between thread creation and locking. The main thread holds a lock while spawning the second thread and only prints the variable before releasing the lock. However, the second thread needs to use the lock before writing 42 to the variable. In the third example the write of 42 would need to be scheduled between the assignment x = 17 and the print which is imposible due to the shared lock. The fourth example is similar from a scheduling point of view but there are two transfers involved. While both transfers are feasible, there is no run exhibiting both. The last example is more involved. Either there is an intervening kill or the program reaches a deadlock. A lock-sensitive DPN-based analysis can treat all these effects precisely.

2 The Monitor-DPN Model

DPNs precisely model unbounded recursion and thread creation based on finite abstractions of method local and thread state. Intuitively a DPN is a set of push down systems which are able to add new push down systems as a side effect of their transitions. In a Monitor-DPN we also allow the threads to communicate via a finite set of reentrant locks which are used in a well-nested fashion. Reentrance means that a thread may acquire the same lock multiple times and releases it only after the matching number of release-operations. We enforce well-nestedness syntactically by allowing locks only to be acquired when pushing a local state to the stack and releasing it implicitly when the old stack level is reached again. That is, lock acquisition and release is bound to procedure calls.

Definition 1. A *Monitor-DPN* $\mathscr{M}$ is a tuple $(\mathrm{Act}, P, \Gamma, X, \Delta, (p_0, \gamma_0))$ consisting of an initial configuration $(p_0, \gamma_0) \in P \times \Gamma$ and finite sets of: actions Act, control states P, stack symbols Γ, locks X and a set Δ of transformation rules of the form:

$$\text{(Base) } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \quad \text{(Call) } p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma_r \quad \text{(Return) } p\gamma \overset{a}{\hookrightarrow} p'$$
$$\text{(Spawn) } p\gamma \overset{a}{\hookrightarrow} p_s\gamma_s p'\gamma' \quad \text{(Monitor) } p\gamma \overset{a,x}{\longrightarrow} p'\gamma'\gamma_r$$

where $p, p', p_s \in P$, $a \in \mathrm{Act}$, $x \in X$, and $\gamma, \gamma', \gamma_r, \gamma_s \in \Gamma$. An *ordinary DPN* is a Monitor-DPN without monitor rules.

We fix a Monitor-DPN $\mathscr{M} = (\mathrm{Act}, P, \Gamma, X, \Delta, (p_0, \gamma_0))$. In order to define a semantics for (Monitor-)DPNs, we assume that there is an infinite set T of thread identifiers with a fixed initial identifier $t_0 \in T$. Moreover, we extend the set of locks by a special symbol to $\bar{X} = X \uplus \{✠\}$, where we write $\uplus$ for disjoint union. We assume that the sets P, Γ, X, Δ, and T are pairwise disjoint.

Definition 2. A thread configuration is a word from $P(\Gamma\bar{X})^*$. A DPN configuration is a finite set of pairs of thread identifiers and thread configurations $c \in \mathscr{C}_{\mathscr{M}} = 2^{T \times P(\Gamma\bar{X})^*}$. The initial configuration is $\{(t_0, p_0\gamma_0✠)\}$. The set of locks *held* by a DPN configuration c is $l(c) = \{x \in X \mid \exists t, w, w'.\ (t, wxw') \in c\}$. The transition rules are as follows:

$$
\begin{array}{lll}
c \uplus \{(t, p\gamma xw)\} & \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'\gamma'}_t & c \uplus \{(t, p'\gamma' xw)\} \\
c \uplus \{(t, p\gamma yw\} & \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'}_t & c \uplus \{(t, p'w)\} \\
c \uplus \{(t, p\gamma yw)\} & \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma_r}_t & c \uplus \{(t, p'\gamma'✠\gamma_r yw)\} \\
c \uplus \{(t, p\gamma yw)\} & \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p_s\gamma_s p'\gamma'}_t & c \uplus \{(t, p'\gamma' yw), (t_s, p_s\gamma_s✠)\} \quad t_s \text{ fresh} \\
c \uplus \{(t, p\gamma yw)\} & \xrightarrow{p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma_r}_t & c \uplus \{(t, p\gamma' x\gamma_r yw)\} \quad \text{if } x \notin l(c)\ .
\end{array}
$$

The lock-insensitive semantics for DPN is obtained by dropping the constraint $x \notin l(c)$. The set $\Pi(\mathscr{M})$ of executions of a DPN consists of all sequences of the form $\{(t_0, p_0\gamma_0✠)\} \xrightarrow{\eta_1}_{t_1} c_1 \ldots \xrightarrow{\eta_n}_{t_n} c_n$.

Execution Trees

We now construct the trees on which our actual analysis is based. Given an execution π of a DPN $\mathscr{M}$ we identify each transition η with it's position in the trace such that π can be seen as a sequence of distinguishable unique operations. The projection of π to the operations performed by thread t is denoted as π^t. These subtraces correspond to traces of ordinary push down systems and their class is therefore equivalent to that of context free languages. Similar to the insight that the set of parsing trees of a context free grammar is tree-regular, we can transform the traces π^t into trees $人(\pi^t)$. This is done by matching return transitions to corresponding calls. For convenience, we add a special node $p\gamma$ to the end of π^t, if the configuration reached by t has a non empty stack, where p is the reached control state and γ is the top of stack. Let $\pi^t = \eta_0 \ldots \eta_i \eta_{i+1} \ldots \eta_j \eta_{j+1} \eta_k$ and let η_i be the first call in π^t matched by a return, say η_j. Then we transform π^t to the tree $\eta_0 \ldots \eta_i(\, \eta_{i+1} \ldots \eta_j\, ,\, \eta_{j+1} \ldots \eta_k\,)$ and continue recursively with the new subtraces. As usual we represent trees by terms. By recursively *hooking in* the $人(\pi^t)$ trees, for all threads but the main thread, as left children of the η nodes which spawned them we obtain what Gawlitza et al. [4] called an execution tree also denoted by $人(\pi)$. By convention the execution tree corresponding to the empty trace is the initial node $p_0\gamma_0$. We define the lock sensitive successors of a set A of execution trees as $lspost^*_{\mathscr{M}}(A) = \{人(\pi\pi') \mid \pi, \pi\pi' \in \Pi(\mathscr{M}),\ 人(\pi) \in A\}$. Moreover the lock-insensitive version $post^*_{\mathscr{M}}$ is the one corresponding to the lock-insensitive version of the semantics. Note that different traces may produce the same execution tree as the tree does not capture the relative order of the transitions of different threads completely. An execution that produces a given execution tree is called a schedule. The configuration reached by an execution can easily be reconstructed from an execution tree up to the naming of threads. Alternatively one can interpret execution trees as a form of decorated configurations.

We introduce additional notations for execution trees: We denote a node which corresponds to a base transition η as BASE^η. An unary call node is denoted as NCALL^η a binary one as RCALL^η. RET^η denotes a return leaf and NIL^p_γ a $p\gamma$ leaf. Monitor transitions are denoted like calls as $\mathrm{ACQ}^\eta_{x,r}$ and $\mathrm{USE}^\eta_{x,r}$

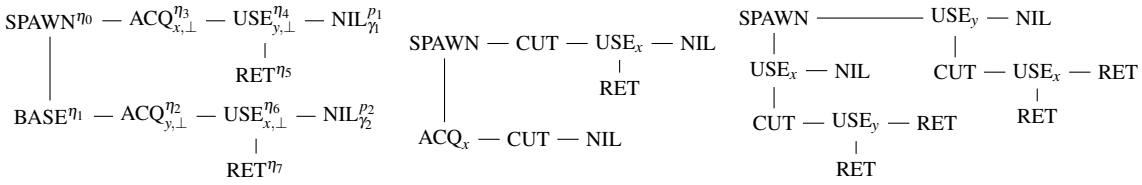


Figure 1: A not lock-sensitively schedulable execution tree and two execution trees with cut.

where we assume additional annotations of the used lock x and a bit $r \in \mathbb{B} = \{\bot, \top\}$ (interpreted as *false* and *true*) indicating whether the operation was reentrant.

An example of an execution tree reachable only in the lock-insensitive semantics is given in Figure 1. The left child of each branch grows downwards. One thread wants to acquire x and then use y. The other thread wants to acquire y and then use x. This is not possible as one would unavoidably reach a deadlock.

Gawlitza et al. [4] showed that the set of reachable execution trees for DPN is effectively tree-regular. We pick up this approach and extend it to reentrant locks. After introducing the concept of a *cut* of an execution tree, we use acquisition and release structures [8] to also obtain an iterable tree-regular characterization of reachable configurations. All this is done by defining appropriate finite tree automata.

A finite tree automaton $\mathscr{T}$ over a ranked alphabet Σ is a triple (Q, Q_f, δ) consisting of finite sets of states Q, accepting states $Q_f \subseteq Q$, and rules δ of the form $f\ q_1 \dots q_n \to q$, where $f \in \Sigma$ is a symbol of rank n and $q, q_1, \dots, q_n \in Q$. If a tree automaton can recognize the trees $t_1, \dots, t_n$ with states $q_1, \dots, q_n$ respectively and it contains the above rule, it can recognize the tree $f(t_1, \dots, t_n)$ with state q. It accepts a tree if it can be recognized with an accepting state. The language $\mathscr{L}(\mathscr{T})$ is the set of all trees accepted by the automaton. A set of trees is called tree-regular if it is the language of a finite tree automaton. The intersection of tree-regular sets is again tree-regular (accepted by the product automaton) and emptiness of the language of a given tree automaton is decidable in linear time. For an introduction to tree automata we refer to Comon et al. [2]. When defining concrete automata we leave out sub- and superscripts of nodes which do not influence the automata and write an underscore to denote arbitrary values, e.g., $\mathrm{NIL} \to \_$ denotes $\mathrm{NIL}^p_\gamma \to U$ for all p, γ and U.

Definition 3. The lock-insensitive execution trees of a Monitor-DPN $\mathscr{M}$ are accepted by the tree automaton $\mathscr{T}_\mathscr{M}$. The state space is $P \times \Gamma \times P \times \{N,T\} \times 2^X$ and the accepting states are $\{p_0\} \times \{\gamma_0\} \times P \times \{N,T\} \times \{\emptyset\}$. For all $\eta \in \Delta$, $t \in \{N,T\}$, $p \in P$, $\gamma \in \Gamma$ and $ls \subseteq X$ the automaton contains the following rules. We write $\frac{f\ q_1 \dots q_n}{q}$ to denote the rule $f\ q_1 \dots q_n \to q$.

$$\frac{\mathrm{NIL}^p_\gamma}{(p,\gamma,p,N,ls)}, \qquad \frac{\mathrm{RET}^\eta}{(p,\gamma,p',T,ls)} \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p',$$

$$\frac{\mathrm{BASE}^\eta\ (p',\gamma',p'',t,ls)}{(p,\gamma,p'',t,ls)} \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma', \qquad \frac{\mathrm{RCALL}^\eta\ (p',\gamma',p'',T,ls)\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)} \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma'',$$

$$\frac{\mathrm{NCALL}^\eta\ (p',\gamma',p'',N,ls)}{(p,\gamma,p'',N,ls)} \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma', \qquad \frac{\mathrm{SPAWN}^\eta\ (p',\gamma',\_,\_,\emptyset)\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)} \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma p''\gamma'',$$

$$\frac{\mathrm{ACQ}^\eta_{x,r}\ (p',\gamma',p'',N,ls\cup\{x\})}{(p,\gamma,p'',N,ls)}, \frac{\mathrm{USE}^\eta_{x,r}\ (p',\gamma',p'',T,ls\cup\{x\})\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)} \quad \text{if } \eta = p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma',\ r = \top \Leftrightarrow x \in ls.$$

Proposition 4. $\mathscr{L}(\mathscr{T}_\mathscr{M}) = post^*_\mathscr{M}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$

Lock-Sensitive Analysis

As $lspost^*_\mathscr{M}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\}) \subseteq post^*_\mathscr{M}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ one can obtain the set of lock-sensitively reachable execution trees by filtering out those which do not have a lock-sensitive schedule. To do so we check the acquisition

structure restricted to non-reentrant actions. The acquisition structure precisely models the restrictions to a possible schedule imposed by the final acquisitions and usages of locks. The restriction to the non-reentrant actions is safe because a reentrant action can always be executed as no other process may hold the lock and it does not introduce new restrictions to other threads due to well-nestedness.

Definition 5. From the set of lock-insensitively reachable execution trees the following tree automaton $\mathscr{T}_{ah}$ accepts those which possess a lock-sensitive schedule. The state space is $2^X \times 2^X \times 2^{X\times X}$ with accepting states $2^X \times 2^X \times \{G \in 2^{X\times X} \mid G \text{ is acyclic}\}$. We write sets of nodes to denote a rule for each node of the given set. The rules are as follows for all $x \in X$:

$$\text{NIL} \to (\emptyset,\emptyset,\emptyset) \quad \text{RET} \to (\emptyset,\emptyset,\emptyset) \quad \text{BASE } \alpha \to \alpha \quad \text{NCALL } \alpha \to \alpha \quad \text{ACQ}_{x,\top}\ \alpha \to \alpha$$
$$\{\text{RCALL}, \text{USE}_{x,\top}, \text{SPAWN}\}\ (A,U,G)\ (A',U',G') \to (A\cup A',\ U\cup U',\ G\cup G') \text{ if } A\cap A' = \emptyset$$
$$\text{USE}_{x,\bot}\ (A,U,G)\ (A',U',G') \to (A\cup A',\ U\cup U'\cup\{x\},\ G\cup G') \text{ if } A\cap A' = \emptyset$$
$$\text{ACQ}_{x,\bot}\ (A,U,G) \to (A\cup\{x\},\ U\cup\{x\},\ G\cup\{(x,u)|u\in U\}) \text{ if } x\notin A$$

Proposition 6. $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah}) = lspost^*_{\mathscr{M}}(\{\text{NIL}^{p_0}_{\gamma_0}\})$

Example 7. A simple application is calculation of may-happen-in-parallel information. That is, given two sets R and W of stack symbols: Is it possible that two threads reach simultaneously a topmost stack symbol of the corresponding sets? The relevant execution trees are accepted by the tree automata $\mathscr{T}_{rw}$ with state space $2^{\{r,w\}}$, single accepting state $\{r,w\}$ and the following rules:

$$\text{NIL}_\gamma \to \{r\} \text{ for } \gamma\in R \qquad \text{NIL}_\gamma \to \{w\} \text{ for } \gamma\in W \qquad \text{NIL}_\gamma \to \emptyset \text{ for } \gamma\notin(R\cup W) \qquad \text{RET} \to \emptyset$$
$$\{\text{ACQ}, \text{BASE}, \text{NCALL}\}\ \alpha \to \alpha \qquad \{\text{RCALL}, \text{SPAWN}, \text{USE}\}\ \alpha\ \beta \to \alpha\cup\beta$$

By checking whether $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah}) \cap \mathscr{L}(\mathscr{T}_{rw}) \stackrel{?}{=} \emptyset$ one can prove absence of a data race on a variable if R represents the set of stack symbols where the variable is read or written and W the set of stack symbols where it is written. This rules out all spurious data races in the examples in Table 1.

Iterable Analysis

We extend this technique to answer reachability questions starting from arbitrary tree-regular sets of reachable execution trees. We exploit the fact that all execution trees of prefixes of an execution are closely related to prefixes of the final tree: They are obtained by cutting the final tree, replacing subtrees by NIL-nodes and turning returning calls or usages into returning calls or acquisitions as necessary. Note however, that not all prefixes of an execution tree correspond to prefixes of its schedules. We use an additional type of node (a CUT-node) to *mark* an intermediate configuration in the tree. Firstly one needs to check that these nodes actually mark an intermediate configuration, we say the tree is cut-wellformed. This can be done by a tree automaton ($\mathscr{T}_{cwf}$). In a second step one can use a tree-transducer (a tree automaton with output) to obtain the execution tree corresponding to the configuration marked by the cut ($\mathscr{T}_{ct}$). Note that the inverse image of tree-regular sets under a tree transducer is effectively tree-regular.

Proposition 8. *For tree-regular* $A \subseteq post^*_{\mathscr{M}}(\{\text{NIL}^{p_0}_{\gamma_0}\})$ *it holds that*
$post^*_{\mathscr{M}}(A) = (\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{cwf}) \cap \mathscr{T}_{ct}^{-1}(A))|_{\text{CUT}}$

Here, $|_{\text{CUT}}$ denotes the projection which removes all cut nodes from the tree which can be done by a simple tree transducer.

Iterable Lock-Sensitive Analysis

In the lock-sensitive case, it remains to check whether the final configuration can lock-sensitively be reached from the intermediate configuration marked by the cut. Two prototypical examples where this

is not possible are shown in the right of Figure 1. To obtain a sufficient criterion we additionally use so called release structures which, analogously to acquisition structures for acquisitions, model restrictions for releasing held locks [8] . Firstly one must ensure that all locks used at the cut can actually be released afterwards (right part of Figure 1). Secondly one must check that no lock finally acquired before the cut is used by another thread after the cut (center part of Figure 1).

Proposition 9. *There exists a tree-automata $\mathscr{T}_{rs}$ such that for tree-regular $A \subseteq lspost^*_{\mathscr{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ $lspost^*_{\mathscr{M}}(A) = (\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{rs}) \cap \mathscr{L}(\mathscr{T}_{cwf}) \cap \mathscr{T}_{ct}^{-1}(A))|_{\mathrm{CUT}} \cap \mathscr{L}(\mathscr{T}_{ah})$*

Implementation

We implemented our approach for the analysis of concurrent Java programs. The implementation is based on the T.J. Watson Libraries for Analysis (WALA)[1], which provides control flow and points-to information. Following Esparza and Knoop [3], we encoded the local state in the stack symbols of the DPN, which consist, in the general setting, only of the control point of the program. The control states are used to model the thread state (e.g. thrown exceptions) and return information of synchronized blocks as these are modeled as multi-exit procedures. We translate the generated models to tree automata which are encoded as logical programs for the XSB system[2], a logic programming and deductive database system. The required tree-automata operations like intersection and checking for emptiness can straight forwardly be implemented in this environment.

References

[1] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR 2005*, volume 3653 of *LNCS*. Springer, 2005.

[2] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[3] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FoSSaCS '99*, volume 1578 of *LNCS*. Springer, 1999. ISBN 3-540-65719-3.

[4] Thomas Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *VMCAI 2011*, volume 6538 of *LNCS*. Springer, 2011.

[5] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[6] Vineet Kahlon and Aarti Gupta. An automata-theoretic approach for model checking threads for ltl propert. In *LICS*. IEEE, 2006.

[7] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *CAV*, volume 3576 of *LNCS*. Springer, 2005. ISBN 3-540-27231-3.

[8] Peter Lammich, Markus Müller-Olm, and Alexander Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *CAV 2009*, volume 5643 of *LNCS*. Springer, 2009.

[9] Benedikt Nordhoff, Peter Lammich, and Markus Müller-Olm. Iterable forward reachability analysis of Monitor-DPNs. Submitted for publication, 2012.

[1] `http://wala.sf.net`

[2] `http://xsb.sf.net`

Formal Verification of Agents Learning by Reinforcement *

Shashank Pathak
IIT, Genova & UniGe
Genova, Italy
shashank.pathak@iit.it

Giorgio Metta
IIT, Genova & UniGe
Genova, Italy
Giorgio.Metta@iit.it

Luca Pulina
Università degli Studi di Sassari
Sassari – Italy
lpulina@uniss.it

Armando Tacchella
UniGe
Genova, Italy
Armando.Tacchella@unige.it

Abstract

This paper describes our current research on safety assessment of artificial agents learning control policies in an uncertain world. In this study, a robotic manipulator learns to play game of air hockey, through Reinforcement Learning(RL). RL is a widely used paradigm in Artificial Intelligence to implement an adaptive behaviour for an agent in little known environment. Here, agent learns through exercising various actions in different states and through feedbacks received while performing those actions. However, some of these actions might be unsafe for the agent in some states. Firstly, we model game of playing air hockey as a task which involves implicit risks related with high-velocities of robotic arm. Secondly, we show that though it is indeed possible to learn stochastic policy via RL where such unsafe situations are penalized, it is not possible to eliminate such risk altogether. Thirdly, we would strive to repair this policy without sacrificing effectiveness of skill thus learnt. To summarise, we apply formal methods to obtain a general and effective intelligent behaviour from an artificial agent. This is in contrast with other methods, such as domain-specific Lyapunov candidate based control or risk-sensitive rewarding based learning.

1 Learning by Reinforcement

Learning by reinforcement – see, e.g., [11] – is one of the most widely adopted paradigms to obtain intelligent behaviour from agents actively engaged in interactions with a surrounding environment. The basic idea behind this type of learning is that an agent is capable of perceiving the current state of the environment wherein it is situated, as well as a feedback signal that occurs each time the agent acts in the environment. Feedback can be either an immediate reward or penalty, and the ultimate goal of the agent is to find a course of action that is guaranteed to maximize accumulated rewards over time. In this sense, learning by reinforcement – also Reinforcement Learning (RL) – can be seen as a way to synthesize (optimal) control programs for agents whose actual knowledge about the external environment is very limited [12]. RL algorithms have shown robust and efficient learning on a variety of problems, particularly in robotics – see, e.g. [8, 7] – and artificial game playing – see, e.g., [13].

*This is preliminary discussion of current and unpublished work

1.1 Preliminaries of Reinforcement Learning

Essential components of RL are: states, actions, rewards and policy. An agent perceives the current state s_t and using some policy, which is a mapping $\pi(s) \to a$ from state to action, takes an action a_t. As a consequence of this action, environment might change and a reward $r_{t+1} \in \mathbb{R}$ is observed by the agent. Consider a sequence of states and rewards in some finite horizon $t \in (t,T)$ as $s_t, r_{t+1}, s_{t+1}, \ldots r_T, s_T$. Also suppose we discount the rewards in future by a factor γ with $\gamma \in [0,1)$ so that rewards received sooner are preferred over those received later. Then, we have average reward given by

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$$

Now we define, value $V^\pi(s) \in \mathbb{R}$, as expected value of this average-reward R_t where $s_t = s$. Given this, approximating optimum *value*, $V^*(s)$ is achieved through an iterative process

$$V(s_t) \to V(s_t) + \alpha(R_t - V(s_t))$$

where $\alpha \in \mathbb{R}$ is a step-size parameter. In order to avoid calculating R_t, we use the fact that if the algorithm converges, value of state $V(s_{t+1})$ should be accurate estimate of expected return. In other words, $E(R_t) = E(r_{t+1}) + \gamma V(s_{t+1})$ and update rule is:

$$V(s_t) \to V(s_t) + \alpha\delta, \quad \delta = (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{1}$$

The rule can also be applied previously visited state, using a factor called *eligibility traces*. Eligibility trace, e_t decays with factor λ and may also be reset, each time an exploratory action is chosen. Intuitively speaking, *eligibility traces* is an algebraic trick to compensate for model-free nature of this kind of learning. Using *eligibility traces*, rewards are propagated backwards along the states of current trajectory, without any mechanism to explicitly store these states.

Analogous to *value*, we define *quality* $Q(s_t, a_t)$ as expected value:

$$E^\pi(R_t | s_t = s, a_t = a)$$

The benefit of doing so is, that if we have update of form similar to 1:

$$Q(s_t, a_t) \to Q(s_t, a_t) + \alpha\delta, \quad \delta = r_{t+1} + \gamma \arg\max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{2}$$

we can have so called *off-policy* learning, where policy followed during learning is irrelevant to learning optimal Q-values. In some practical scenario, this is desirable since it implies that we could do away with costly operation of policy improvement during each update.

1.2 Case Study

Taking up the work of [6], the task assigned to the learning agents is to play defense in the air hockey game. Air hockey is played by two players. They use round paddles (mallets) to hit a flat round puck across a low-friction table. At each end of the table there is a goal area. The objective of the game is to hit the puck so that it goes into the opponent's goal (offense play), and it does not go into your own goal (defense play). Air hockey has already been explored as a benchmark task in robotics and vision – see, e.g., [2, 3] – because it is fast, demanding, and complex, once the various elements of the physical setup are taken into account. The setup considered in [6] is composed of a robot – an industrial manipulator with a mallet fitted on the end effector – playing against a human opponent. What makes

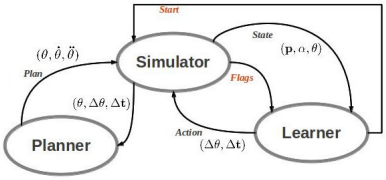


Figure 1: Sketch of the artificial air-hockey setup

this setup interesting also from the verification viewpoint, is that an agent learning to play the game has a non-negligible chance to behave hazardously. For instance, in [6] predicting target positions that are outside the robot's safe operational range is considered unsafe. This is because, given the robot's speed and mass, the energy involved in a collision, e.g., against the table, can easily damage the table and the manipulator. In this study we extend the notion of unsafe behaviour to include the possibility of generating speed profiles which are incompatible with the operating range of the manipulator. Given the kind of game, it is not unlikely that an agent learning by experience tries to observe as long as possible the current trajectory of the puck – to improve prediction of the intercept position – and then move the manipulator as quickly and precisely as possible – to reach the intercept position "right on time".

The task of Learner is to become skilled in defending the robot's goal area across a series of episodes, i.e., shots fired by the opponent. As can be seen in figure 1, the state space is three-dimensional $S = S_1 \times S_2 \times S_3$, where each state $(p, \alpha, \theta) \in S$ encodes the current puck trajectory and robotic arm position. Put differently, (p, α), encode linear trajectory of puck, in perpendicular form while θ encodes angular position of robotic arm. The output action space A is mono-dimensional where each action $\Delta\theta \in A$ defines the suggested angular displacement of the robot's mallet, and we keep time-duration of this action Δt fixed to 0.2 seconds for the sake of simplicity.[1]

In our implementation, we consider RL algorithms in the class of *Temporal Difference* (TD) methods as described before – for details see Chapter 3 of [11]. More specifically, LEARN is an implementation of the *Sarsa*[10] algorithm obtained by defining the auxiliary functions as follows.

- UPDATE is based on the key idea of learning an action-value function instead of a state-value function in order to combine policy improvement with value estimation. In particular, the value of action $a \in A$ performed in state $s \in S$ – denoted as $Q(s,a)$ – is estimated iteratively by the following update equation[2].

$$Q(s_t, a_t) \to Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{3}$$

  where t, γ and α have the same meaning as described before. Notice that, in contrast with (2), *Sarsa* uses Q-values for the next state, $Q(s_{t+1}, a_{t+1})$, and is also an *on-policy* learning method. If the episode ends with s_{t+1}, then $Q(s_{t+1}, a_{t+1})$ is defined to be zero for all actions $a \in A$. Since there are finitely many states $s \in S$ and actions $a \in A$, all the values $Q(s,a)$ can be saved in a finite-sized lookup-table.

- COMPUTEPOLICY selects, in any given state s, the action a^* such that $a^* = \text{argmax}_a Q(s,a)$ most

[1] In our current implementation p and α are quantized over 58 and 37 values, respectively, and θ is quantized over 31 values, yielding 66526 possible states. The number of possible actions $\Delta\theta$ is 31.

[2] The fact that equation (3) uses the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ gives the name to the algorithm.

of the times and, with a small probability ε, it selects a^* at random – this is known as ε-*greedy policy* in the literature [11].

- COMPUTEREWARD works according to two different schemes. The *safety oblivious* scheme considers a baseline value $R \in \mathbb{R}^+$ which is awarded to the agent when it saves a goal. A smaller reward, $\frac{R}{5}$ is given when the puck is hit even if the current trajectory is not leading to a goal. A negative reward (penalty) $-R$ is assigned when the puck hits the robot's goal area. Finally, to speed up learning, a small positive reward $\frac{R}{50}$ is assigned to each non-terminal state. The *safety aware* scheme is identical to the first one, with the exception of a penalty $\frac{-R}{5}$ assigned when the speed-warning flag w becomes true.

Overall algorithm for learning is as described in 1:

Algorithm 1 Pseudo-code for learning to play Air hockey using Reinforcement Learning

```
Initialize Q ← 0; Δt ← 20ms
function LEARN(N_e, N_b, N_r)
    for all i ∈ {1,...,N_e} do
        Send Start signal to Simulator
        j ← 1
        repeat
            Receive s_j ← (p_j, α_j, θ_j) from Simulator
            Δθ_j ← COMPUTEPOLICY(Q, s_j)
            Send (Δθ_j, Δt) to Simulator
            Receive s_{j+1} ← (p_{j+1}, α_{j+1}, θ_{j+1}) and
                    f_{j+1} ← (m, g, w, r)
            r_{j+1} ← COMPUTEREWARD((s_{j+1}, f_{j+1}))
            E_j ← (s_j, Δθ_j, r_{j+1}, s_{j+1}, f_{j+1})
            Q ← UPDATE(Q, E_j)
            j ← j + 1
            if (j = N_b) then
                for all k ∈ {1,...,N_r} do
                    Choose random m ∈ {1,...,N_b}
                    Q ← UPDATE(Q, E_j)
                end for
                j ← 1
            end if
        until r = TRUE
    end for
    return Q
end function
```

2 Verification and Repair

After learning, the agent follows a stochastic policy. To be precise, the policy is a Softmax-distribution [11] over state-action Q-value where both states as well as actions are discrete. Assuming stationarity of the environment, such an agent-environment system could be modeled as a Discrete Time Markov Chain (DTMC), $\mathscr{D}$, where the transition probabilities are assigned by the policy learnt. Overall safety of the policy learnt, could be assessed by analysing frequency of reaching the bad states and probability with which this frequency exceeds a given threshold. Alternatively, safety could be assessed by more stringent requirement of reachability of unsafe states. We have investigated both the approaches and will describe the latter in more detail. In this case, in order to be safe, policy must not allow reachability to unsafe states with probability greater than some threshold probability, P_{bound}. Hence goal of verification is to assert that overall reachability to unsafe states is within a probability bound P_{bound}. In other words,

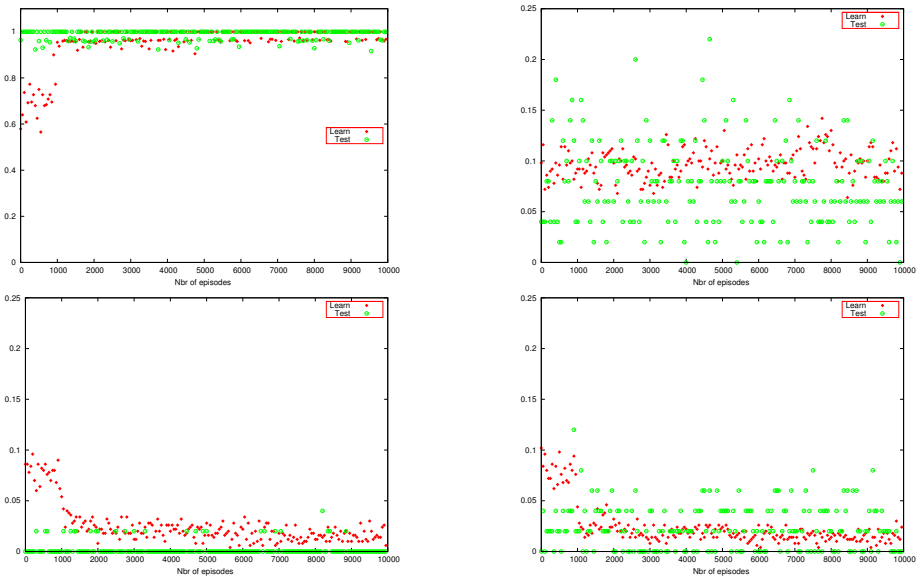


Figure 2: Effectiveness and safety across learning trials: Learning without considering safety (first row), Learning with safety-sensitive rewards (second row); First top-left shows skill of defense-play learned by agent, while top-right shows how this does not make agent safer in both learning and test phase. Bottom-left is when unsafe transitions are treated as faults while bottom-right is when they are treated as failures. The plots visualize performances of the LEARN algorithm over an increasing number of learning episodes N_e reported on the x-axis. For each such number, the plots report percentage of saved goals and percentage of unsafe actions while learning. We also report average percentage of unsafe transitions and of saved goals over test trials.

the property is $\mathscr{P} = P_{<P_{bound}}(\Diamond s = s_{unsafe})$. We used probabilistic model checker PRISM [5] to check the stochastic policy learnt. Our preliminary results, before verification, as seen in figure 2 confirm that, though risk-sensitive rewarding in learning could help agent behave safer, the overall risk to the agent is still quite significant. Here, safety is measured as overall ratio of visit to bad states and skill as ratio of goals saved to total shots in direction of goal. Both these measures are averaged over 50 episodes. There is clearly a tension between learning to acquire a skill and maintaining self-safety. Verification enabled us to confirm this quantitatively.

As mentioned before, through probabilistic model checking, we are able to determine quantitatively the risk inherent in the system. We are currently working on generating counter-examples in efficient as well as automated fashion, using tools such as COMICS [4]. Moreover, we seek to use them to repair the learnt policy so that it meets minimum safety guarantee, set a-priori by physical requirements of the agent. To be precise, a probability bound P_{bound} is imposed on total reachability of bad states in DTMC $\mathscr{D}$.

For this we have approached the problem, seeking to repair the causes of unsafe behaviour of the policy. The root cause of avoidable unsafe behaviour, lies in action selection of the agent. *Action-selection* in a learning scenario has been explored more in cognitive science and psychology, for example [9], and there was a recent EuCogII conference dedicated to it [1]. We are however, interested in quantitative probabilistic nature of this action selection, a part of which might lead to unsafe states with overall probability higher than P_{bound}. In a more specific setting of *Sarsa* learning and associated DTMC $\mathscr{D}$ that we verify, this amounts to changing the policy learnt. While changing the policy too much might invalidate learning, changing it too little might not make it safe enough. In addition to this, making a significant local change to policy might also destabilize learning. Hence, to solve this problem in a principled and generalised way, we are working on an approach Repair, inspired from *eligibility traces*, where we grad-

ually penalise a state based on its proximity to a bad state. The pseudo-code for such a repair is given in 2. Here, P_{unsafe} is obtained from $\mathscr{D}$, using a model checker, and $\mathscr{D}$ is repaired until it no longer negates property $\mathscr{P}$ with bound P_{unsafe}. At this point, model checker is used again, to generate new P_{unsafe} and $\mathscr{D}$ is repaired again. This continues till the system satisfies $\mathscr{P}$ with bound P_{bound}.

Algorithm 2 Pseudo-code for Verification and Repair of Learn

Given agent $\mathscr{A}$, learning algorithm Learn 1 , safety bound P_{bound}
Using $\mathscr{A}$ perform Learn
Obtain policy $\pi(s,a)$
Construct a DTMC $\mathscr{D}$ from policy $\pi(s,a)$
Use MRMC or PRISM on $\mathscr{D}$ to obtain P_{unsafe} of violating $\mathscr{P}$
repeat
  repeat
    Use COMICS to generate set of all paths $\mathbf{S}_{unsafe}$ negating $\mathscr{P}$ with bound P_{unsafe}
    Apply Repair on $\mathbf{S}_{unsafe}$
  until $\mathbf{S}_{unsafe} = \{\phi\}$
  $P_{unsafe} \leftarrow P_{unsafe} - \varepsilon$, $\varepsilon \in (0, P_{unsafe} - P_{bound}]$
until $P_{unsafe} < P_{bound}$

References

[1] Action-Selection. Fifth members conference of european network for the advancement of artificial cognitive systems, interaction and robotics. `http://www.eucognition.org/index.php?page=fifth-conference-general-info`, October 2011.

[2] D. C. Bentivegna, C. G. Atkeson A. Ude, and G. Cheng. Learning to Act from Observation and Practice. *International Journal of Humanoid Robotics*, 1(4), December 2004.

[3] BE Bishop and MW Spong. Vision-based control of an air hockey playing robot. *IEEE Control Systems Magazine*, 19(3):23–32, 1999.

[4] N. Jansen, E. Ábrahám, M. Scheffler, M. Volk, A. Vorpahl, R. Wimmer, J. Katoen, and B. Becker. The comics tool - computing minimal counterexamples for discrete-time markov chains. *CoRR*, abs/1206.0603, 2012.

[5] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001.

[6] G. Metta, L. Natale, S. Pathak, L. Pulina, and A. Tacchella. Safe and effective learning: A case study. In *ICRA*, pages 4809–4814, 2010.

[7] A. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, pages 363–372, 2006.

[8] J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pages 1–20, 2003.

[9] K.R. Ridderinkhof, W.P.M. van den Wildenberg, S.J. Segalowitz, C.S. Carter, et al. Neurocognitive mechanisms of cognitive control: the role of prefrontal cortex in action selection, response inhibition, performance monitoring, and reward-based learning. *Brain and cognition*, 56(2):129–140, 2004.

[10] G.A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[11] R.S. Sutton and A.G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.

[12] R.S. Sutton, A.G. Barto, and R.J. Williams. Reinforcement learning is direct adaptive optimal control. *Control Systems, IEEE*, 12(2):19–22, 1992.

[13] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Recent Results and Future Directions in Strategy Logic

Giuseppe Perelli
Università degli studi di Napoli "Federico II"
perelli.gi@gmail.com

Joint work with
Fabio Mogavero[1], Aniello Murano [1], and Moshe Y. Vardi [2].

1 Extended Abstract

In open-system verification, a fundamental area of research is the study of modal logics for strategic reasoning [1, 7, 10]. An important contribution in this field has been the development of *Alternating-Time Temporal Logic* (ATL*, for short), introduced by Alur, Henzinger, and Kupferman [1]. Formally, it is obtained as a generalization of the logic CTL* [8], where the path quantifiers *there exists* "E" and *for all* "A" are replaced with strategic modalities of the form "$\langle\langle$A$\rangle\rangle$" and "[[A]]", for a set A of *agents*. These modalities are used to express cooperation and competition among agents in order to achieve a temporal goal. Several decision problems have been investigated about ATL*; both its model-checking and satisfiability problems are decidable in 2EXPTIME [23], just like they are for CTL*.

Despite its powerful expressiveness, ATL* suffers from two strong limitations: 1) strategies are treated only implicitly through modalities that refer to games between competing coalitions and 2) strategic modalities represent coupled $\exists\forall$ and $\forall\exists$ quantifications over strategies. To overcome this problem, Chatterjee, Henzinger, and Piterman introduced *Strategy Logic* (CHP-SL, for short) [6],which treats strategies in *two-player turn-based games* as *first-order objects*. The explicit treatment of strategies makes this logic very useful and more expressive than ATL*, however, it still suffers from severe limitations. In particular, it is limited to two-player turn-based games and does not allow different players to share the same strategy, suggesting that strategies have yet to become truly first-class objects in this logic. For example, it is impossible to describe the classic strategy-stealing argument of many real-life combinatorial games.

These considerations has led us to introduce and investigate a new *Strategy Logic*, denoted SL, as a more general framework than CHP-SL, for explicit reasoning about strategies in multi-agent concurrent games [14]. Syntactically, SL extends the logic LTL [18] by means of *strategy quantifiers*, the existential $\langle\langle x\rangle\rangle$ and the universal $[[x]]$, as well as *agent binding* (a,x), where a is an agent and x a variable. Intuitively, these elements can be read as *"there exists a strategy x"*, *"for all strategies x"*, and *"bind agent a to the strategy associated with x"*, respectively.

The price that one has to pay for the expressiveness of SL w.r.t. ATL* is the lack of important model-theoretic properties and an increased complexity of related decision problems. In particular, in [13, 14], it was shown that SL does not have the bounded-tree model property and the satisfiability problem is *highly undecidable*, precisely, Σ^1_1-HARD. Moreover, in [12], it was shown that the model checking problem is nonelementary-complete (we recall that also for CHP-SL it is known to be nonelementary, while

[1] Università degli studi di Napoli "Federico II"
[2] Rice University, Houston, Texas, USA

it is open the question whether it is decidable).

The negative complexity results on the decision problems of SL with respect to ATL*, provide motivations for an investigation of decidable fragments of SL, strictly subsuming ATL*, with a better complexity. In particular, by means of these sublogics, one may understand why SL is computationally more difficult than ATL*.

The main fragments we have investigated and studied are *Nested-Goal*, *Boolean-Goal*, and *One-Goal Strategy Logic*, respectively denoted by SL[NG], SL[BG], and SL[1G]. They encompass formulas in a special prenex normal form having nested temporal goals, Boolean combinations of goals, and a single goal at a time, respectively. For goal we mean an SL formula of the type $\flat\psi$, where $\flat$ is a binding prefix of the form $(\alpha_1,x_1),\ldots,(\alpha_n,x_n)$ containing all the involved agents and ψ is an agent-full formula. In SL[1G], each temporal formula ψ is prefixed by a quantification-binding prefix $\wp\flat$ that quantifies over a tuple of strategies and binds them to all agents.

As main results about these fragments, we have proved that the satisfiability and model-checking problems for SL[1G] are 2EXPTIME-COMPLETE, thus not harder than the one for ATL*. On the contrary, for SL[NG], the model checking problem is nonelementary and the satisfiability is undecidable. Finally, we observe that SL[BG] includes CHP-SL, the relative model-checking problem relies between 2EXPTIME and NONELEMENTARYTIME, while the satisfiability problem is undecidable.

To achieve all positive results about SL[1G], we use a fundamental property of the semantics of this logic, called *elementariness*, which allows us to strongly simplify the reasoning about strategies by reducing it to a step-by-step reasoning about which action to perform. This intrinsic characteristic of SL[1G], which unfortunately is not shared by the other two fragments, asserts that, in a determined history of the play, the value of an existential quantified strategy depends only on the values of strategies, from which the first depends, on the same history. This means that, to choose an existential strategy, we do not need to know the entire structure of universal strategies, as for SL, but only their values on the histories of interest.

By means of elementariness, we can solve the SL[1G] decision problems via alternating tree automata in such a way that we avoid the projection operations by using a dedicated automaton that makes an action quantification for each node of the tree model. As this automaton is only exponential in the size of the formula (and independent from its alternation number) and its nonemptiness can be computed in exponential time, we get that both model-checking and satisfiability for SL[1G] are 2EXPTIME. Clearly, the elementariness property also holds for ATL*, as it is included in SL[1G]. In particular, although it has not been explicitly stated, this property is crucial for most of the results achieved in literature about ATL* by means of automata.

All the results reported in this paper come from [13, 12, 14]. The interested reader can refer to these works to find more motivations, examples and related material.

1.1 Informal definitions and examples

Due to lack of space, we report here only the informal definitions of the syntax and the semantic framework of Strategy Logic. For a complete and more precise treatment, we strongly recommend to refer to [13].

SL syntactically extends LTL by means of two *strategy quantifiers*, the existential $\langle\langle x\rangle\rangle$ and the universal $[[x]]$, and *agent binding* (a,x), where a is an agent and x is a variable. Intuitively, these new elements can be respectively read as *"there exists a strategy x"*, *"for all strategies x"*, and *"bind agent a to the strategy associated with the variable x"*. SL *formulas* are built inductively from the sets of atomic propositions AP, variables Var, and agents Ag, by using the following grammar, where $p \in$ AP, $x \in$ Var,

and $a \in \mathrm{Ag}$:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi \mid \langle\langle x\rangle\rangle\varphi \mid [[x]]\varphi \mid (a,x)\varphi.$$

SL denotes the infinite set of formulas generated by the above rules.

In order to practice with the syntax of our logic and express game-theoretic concepts through formulas, we describe two examples of important properties that are possible to write in SL, but neither in ATL* nor in CHP-SL. The first concept we introduce is the well-known deterministic concurrent multi-player *Nash equilibrium* for Boolean valued payoffs.

Example 1 (Nash Equilibrium)**.** Consider the n agents $\alpha_1,\ldots,\alpha_n$ of a game, each of them having, respectively, a possibly different temporal goal described by one of the LTL formulas $\psi_1,\ldots,\psi_n$. Then, we can express the existence of a strategy profile $(\mathsf{x}_1,\ldots,\mathsf{x}_n)$ that is a *Nash equilibrium* (NE, for short) for $\alpha_1,\ldots,\alpha_n$ w.r.t. $\psi_1,\ldots,\psi_n$ by using the SL sentence $\varphi_{NE} \triangleq \langle\langle \mathsf{x}_1\rangle\rangle(\alpha_1,\mathsf{x}_1)\cdots\langle\langle \mathsf{x}_n\rangle\rangle(\alpha_n,\mathsf{x}_n)\,\psi_{NE}$, where $\psi_{NE} \triangleq \bigwedge_{i=1}^{n}(\langle\langle \mathsf{y}\rangle\rangle(\alpha_i,\mathsf{y})\psi_i) \rightarrow \psi_i$. Informally, this asserts that every agent α_i has x_i as one of the best strategy w.r.t. the goal ψ_i, once all the other strategies of the remaining agents α_j, with $j \neq i$, have been fixed to x_j. Note that here we are only considering equilibria under deterministic strategies.

As in physics, also in game theory an equilibrium is not always stable. Indeed, there are games having Nash equilibria that are instable. One of the simplest concepts of stability that is possible to think is called *stability profile*.

Example 2 (Stability Profile)**.** Think about the same situation of the above example on NE. Then, a *stability profile* (SP, for short) is a strategy profile $(\mathsf{x}_1,\ldots,\mathsf{x}_n)$ for $\alpha_1,\ldots,\alpha_n$ w.r.t. $\psi_1,\ldots,\psi_n$ such that there is no agent α_i that can choose a different strategy from x_i without changing its own payoff and penalizing the payoff of another agent α_j, with $j \neq i$. To represent the existence of such a profile, we can use the SL sentence $\varphi_{SP} \triangleq \langle\langle \mathsf{x}_1\rangle\rangle(\alpha_1,\mathsf{x}_1)\cdots\langle\langle \mathsf{x}_n\rangle\rangle(\alpha_n,\mathsf{x}_n)\,\psi_{SP}$, where $\psi_{SP} \triangleq \bigwedge_{i,j=1,i\neq j}^{n}\psi_j \rightarrow [[\mathsf{y}]]((\psi_i \leftrightarrow (\alpha_i,\mathsf{y})\psi_i) \rightarrow (\alpha_i,\mathsf{y})\psi_j)$. Informally, with the ψ_{SP} subformula, we assert that, if α_j is able to achieve his goal ψ_j, all strategies y of α_i that left unchanged the payoff related to ψ_i, also let α_j to maintain his achieved goal. At this point, it is very easy to ensure the existence of an NE that is also an SP, by using the SL sentence $\varphi_{SNE} \triangleq \langle\langle \mathsf{x}_1\rangle\rangle(\alpha_1,\mathsf{x}_1)\cdots\langle\langle \mathsf{x}_n\rangle\rangle(\alpha_n,\mathsf{x}_n)\,\psi_{SP} \wedge \psi_{NE}$.

As semantic framework for our logic language, we use a *graph-based model* for *multi-player games* named *concurrent game structure* [1]. Intuitively, this mathematical formalism provides a generalization of *Kripke structures* and *labeled transition systems*, modeling *multi-agent systems* viewed as games, in which players perform *concurrent actions* chosen strategically as a function on the history of the play.

A *concurrent game structure* (CGS, for short) is a tuple $\mathscr{G} \triangleq \langle \mathrm{AP},\mathrm{Ag},\mathrm{Ac},\mathrm{St},\lambda,\tau,s_0\rangle$, where we respectively have that AP and Ag are finite non-empty sets of *atomic propositions* and *agents*, Ac and St are enumerable non-empty sets of *actions* and *states*, $s_0 \in \mathrm{St}$ is a designated *initial state*, and $\lambda : \mathrm{St} \rightarrow 2^{\mathrm{AP}}$ is a *labeling function* that maps each state to the set of atomic propositions true in that state. Let $\mathrm{Dc} \triangleq \mathrm{Ac}^{\mathrm{Ag}}$ be the set of *decisions*, i.e., functions from Ag to Ac representing the choices of an action for each agent. Then, $\tau : \mathrm{St}\times\mathrm{Dc} \rightarrow \mathrm{St}$ is a *transition function* mapping a pair of a state and a decision to a state.

A *track* (resp., *path*) in a CGS $\mathscr{G}$ is a finite (resp., an infinite) sequence of states $\rho \in \mathrm{St}^+$ (resp., $\pi \in \mathrm{St}^\omega$) such that, for all $i \in [0,|\rho|-1[$ (resp., $i \in \mathbb{N}$), there exists a decision $\mathsf{d} \in \mathrm{Dc}$ such that $(\rho)_{i+1} = \tau((\rho)_i,\mathsf{d})$ (resp., $(\pi)_{i+1} = \tau((\pi)_i,\mathsf{d})$). [3] The set $\mathrm{Trk} \subseteq \mathrm{St}^+$ (resp., $\mathrm{Pth} \subseteq \mathrm{St}^\omega$) contains all tracks (resp., paths). Moreover, $\mathrm{Trk}(s)$ (resp., $\mathrm{Pth}(s)$) indicates the subsets of tracks (resp., paths) starting at a state $s \in \mathrm{St}$.

[3] The notation $(w)_i \in \Sigma$ indicates the *element* of index $i \in [0,|w|[$ of a non-empty sequence $w \in \Sigma^\infty$.

A *strategy* in a CGS $\mathscr{G}$ is a function $\mathsf{f} : \mathrm{Trk} \to \mathrm{Ac}$ that maps each track to an action. The set of all strategies is denoted by Str.

An *assignment* in a CGS $\mathscr{G}$ is a function $\chi : \mathrm{Var} \cup \mathrm{Ag} \to \mathrm{Str}$ that maps variables in a given set Var and agents to the set of strategies. The set of assignments over a certain set of variables Var is denoted by $\mathrm{Asg}(\mathrm{Var})$.

Given a CGS $\mathscr{G}$, for all SL formulas φ, states $s \in \mathrm{St}$, and assignments $\chi \in \mathrm{Asg}(\mathrm{Var})$, with Var be the set of variables occurring in φ, the modeling relation $\mathscr{G}, \chi, s \models \varphi$ is inductively defined as follows.

1. If φ is an atomic proposition or its principal scope is a Boolean or temporal operator, the semantic is defined as usual in LTL.
2. For a variable $x \in \mathrm{Var}$ and a formula φ, it holds that:
   (a) $\mathscr{G}, \chi, s \models \langle\langle x \rangle\rangle \varphi$ if there exists a strategy $\mathsf{f} \in \mathrm{Str}$ such that $\mathscr{G}, \chi[x \mapsto \mathsf{f}]$ [4], $s \models \varphi$;
   (b) $\mathscr{G}, \chi, s \models [\![x]\!] \varphi$ if for all strategies $\mathsf{f} \in \mathrm{Str}$ it holds that $\mathscr{G}, \chi[x \mapsto \mathsf{f}], s \models \varphi$.
3. For an agent $a \in \mathrm{Ag}$, a variable $x \in \mathrm{Var}$, and a formula φ, it holds that $\mathscr{G}, \chi, s \models (a,x)\varphi$ if $\mathscr{G}, \chi[a \mapsto \chi(x)], s \models \varphi$.

Intuitively, at Items 2a and 2b, respectively, we evaluate the existential $\langle\langle x \rangle\rangle$ and universal $[\![x]\!]$ quantifiers over strategies, by associating them to the variable x. Moreover, at Item 3, by means of an agent binding (a,x), we commit the agent a to a strategy associated with the variable x.

Finally, we say that a CGS $\mathscr{G}$ is a *model* of an SL sentence φ, in symbols $\mathscr{G} \models \varphi$, if $\mathscr{G}, \chi, s_0 \models \varphi$ for all assignments χ. An SL sentence φ is *satisfiable* if there is a model for it.

1.2 Work in Progress and Future Directions

In [14], Strategy Logic has been introduced as a new powerful formalism for reasoning about strategies. There, it has been shown that the satisfiability problem is undecidable. In the extended version [13] of [14], fragments of SL have been introduced and investigated as far as the model-checking problem is concerned. In particular, it turns out that while for SL the model-checking is NONELEMENTARYTIME-COMPLETE, it is 2EXPTIME-COMPLETE for SL[1G] (thus not harder than that for ATL*). The question about SL[BG] is open. In [12], the satisfiability problem for the fragments we have introduced in [12] has been investigated. It turns out that this problem is undecidable for SL[BG] while it remains 2EXPTIME-COMPLETE for SL[1G] (as for ATL*).

Out of the above picture, SL[1G] is the biggest known decidable fragment of SL, strictly subsuming ATL*. On the other side, the bigger (but undecidable) logic SL[BG] is of major interest. Indeed, it can describe several interesting properties non expressible in SL[1G] such as *Nash equilibrium, strong Nash equilibrium, sub-game perfect equilibrium, coalition proof Nash equilibrium*, etc. (for a survey about Nash Equilibrium and the like, see [16, 15]). For these reasons, it is our intention to keep investigating SL[BG]. It is worth noting that SL[BG] is strictly more expressive than CHP-SL, for which the exact complexity of the model-checking problem is still open. So, solving the model-checking problem for SL[BG] would solve the problem for CHP-SL as well. A possible way to attack the model-checking problem is to proceed by steps, introducing some new fragment of SL[BG] (subsuming SL[1G]) and solving their model-checking problem. In this direction, Mogavero, Murano, and Sauro have defined in [11] a fragment in which it is allowed only the conjunction of goals, while the disjunction is avoided This logic is called *Strategy Logic Conjunctive Goal*. They also proved that such a fragment has the elementariness property and then, by applying the procedure explained in [14], its model-checking problem is

[4] By $\chi[x \mapsto \mathsf{f}]$ we are denoting the assignment obtained from χ by substituting only the value of $\chi(x)$ with f.

2EXPTIME-COMPLETE.

Since ATL* and SL[1G] have the same complexities for both the model-checking and satisfiability problems, a natural question that arises is whether to lift the most common extensions of ATL* to SL[1G], in order to retain the model properties and the complexities of the related decision problems.

A very interesting extension in this direction is the use of graded modalities. Such modalities allow to generalize the existential and universal quantifiers to express that, for a natural number n, there exists at least n elements or for all except n elements that satisfy a formula. Several logics have been extended with graded modalities and the complexities of the related decision problems have been deeply investigated. In [9], Kupferman et al. introduced *Graded μ-calculus* to extend μ-calculus with graded modalities. They proved that such extension is exponentially more succinct than μ-calculus, while the satisfiability problem is EXPTIME-COMPLETE, thus not harder than for μ-calculus (for a recent and complete survey of the subject, please read [5]). In [3, 4, 2], Bianco, Mogavero, and Murano introduced *Graded Computation-Tree Logic* (GCTL, for short), a modal logic that extends CTL by replacing the universal (A) and existential (E) quantifiers with their graded versions $\mathsf{A}^{<n}$ and $\mathsf{E}^{\geq n}$. They proved that, despite such extension is strictly more expressive than CTL, the satisfiability problem for GCTL is EXPTIME-COMPLETE, as it is for CTL, even in the case that the graded numbers are coded in binary. Our aim is to investigate the expressive power and the complexities of the classical decision problem for *Graded*-SL (GSL, for short). Precisely, GSL is obtained by replacing the universal ($[\![\mathsf{x}]\!]$) and existential ($\langle\langle\mathsf{x}\rangle\rangle$) strategy quantifiers in SL[1G] with graded modalities of the form $[\![\mathsf{x}]\!]^{<n}$ and $\langle\langle\mathsf{x}\rangle\rangle^{\geq n}$. Informally speaking, these two operators have the meaning of "there exists at least n different non-equivalent strategies ..." and "for all except at most n non-equivalent strategies ..." respectively. Despite this extension is natural and all the reasons introduced in GCTL seem to be easily liftable to this new logic, there is deep work to do regarding the equivalence among strategies. Indeed, to deal with this logic, a suitable concept of *non-equivalent* strategies has to be provided. We expect that GSL is strictly more expressive than SL. We plan to investigate the complexities of both the model-checking and the satisfiability problems for GSL and its fragments and to retain the same complexity results.

Another extension we would like to address is SL with coalition modalities. *Coalition Logic* (CL, for short) has been introduced in [17] for reasoning about games. It can be viewed as the fragment of ATL in which only the next operator, combined with the universal ($[A]$) and existential ($\langle A\rangle$) quantifiers over set of agents, is allowed. In [21, 22], Ågotnes, van der Hoek, and Wooldridge introduced *Quantified Coalition Logic* (QCL, for short) as an extension of CL in which the quantifications over sets of agents are replaced with universal ($[\mathsf{P}]$) and existential ($\langle\mathsf{P}\rangle$) quantifications over properties of coalitions. They have shown that QCL is as expressive as CL but exponentially more succinct. Moreover, they proved that the complexity of model-checking and satisfiability problem for QCL is no worse than the ones for CL. By using a similar idea, we intend to lift to SL and, in particular, to SL[1G] the reasoning about coalition with the aim to find a more succinct representation of formulas without blowing up their complexities.

Last but not least, another possible extension of SL regards the concept of *normative systems* introduced in [24, 25] that are used to reason about coordination of multi-agent systems. In [19, 20], Ågotnes, van der Hoek, Rodriguez-Aguilar, Sierra, and Wooldridge exploited such concept to extend CTL with *Normative Temporal Logic* and proved that its model-checking problem is EXPTIME-HARD. By using the same idea, it is possible to extend also SL and, in particular, SL[1G] with normative systems and study the related decision problems.

References

[1] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. *JACM*, 49(5):672–713, 2002.

[2] A. Bianco, F. Mogavero, and A. Murano. Graded Computation Tree Logic. Association for Computing Machinery. Under publication.

[3] A. Bianco, F. Mogavero, and A. Murano. Graded Computation Tree Logic. In *LICS'09*, pages 342–351. IEEE Computer Society, 2009.

[4] A. Bianco, F. Mogavero, and A. Murano. Graded Computation Tree Logic with Binary Coding. In *CSL'10*, LNCS 6247, pages 125–139. Springer, 2010.

[5] P.A. Bonatti, C. Lutz, A. Murano, and M.Y. Vardi. The Complexity of Enriched Mu-Calculi. *LMCS*, 4(3):1–27, 2008.

[6] K. Chatterjee, T.A. Henzinger, and N. Piterman. Strategy Logic. *IC*, 208(6):677–693, 2010.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2002.

[8] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. *JACM*, 33(1):151–178, 1986.

[9] O. Kupferman, U. Sattler, and M.Y. Vardi. The Complexity of the Graded μ-Calculus. In *CADE'02*, LNCS 2392, pages 423–437. Springer, 2002.

[10] O. Kupferman, M.Y. Vardi, and P. Wolper. Module Checking. *IC*, 164(2):322–344, 2001.

[11] F. Mogavero, A. Murano, and L.Sauro. Again and Again Beyond ATL*. Personal communication, 2012.

[12] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. What makes atl* decidable? a decidable fragment of strategy logic. In *CONCUR*, pages 193–208, 2012.

[13] F. Mogavero, A. Murano, G. Perelli, and M.Y. Vardi. Reasoning About Strategies: On the Model-Checking Problem. Technical Report 1112.6275, arXiv, December 2011.

[14] F. Mogavero, A. Murano, and M.Y. Vardi. Reasoning About Strategies. In *FSTTCS'10*, LIPIcs 8, pages 133–144, 2010.

[15] M.J. Osborne. *An Introduction to Game Theory*. Oxford University Press, 2009.

[16] M.J. Osborne and A. Rubinstein. *A Course in Game Theory.* MIT Press, 1994.

[17] M. Pauly. A Modal Logic for Coalitional Power in Games. *JLC*, 12(1):149–166, 2002.

[18] A. Pnueli. The Temporal Logic of Programs. In *FOCS'77*, pages 46–57, 1977.

[19] T. Ågotnes, W. van der Hoek, C. Sierra J. A. Rodriguez-Aguilar, and M. Wooldridge. On the logic of normative systems. In *IJCAI 2007*, pages 1175–1180, California, 2007. AAAI Press.

[20] T. Ågotnes, W. van der Hoek, C. Sierra J. A. Rodriguez-Aguilar, and M. Wooldridge. A temporal logic of normative systems. In *Towards Mathematical Philosophy*, volume 28 of *Trends in Logic*, pages 69–106. Springer Netherlands, 2009.

[21] T. Ågotnes, W. van der Hoek, and M. Wooldridge. Quantified coalition logic. *Synthese*, 165(2):269–294, 2008.

[22] T. Ågotnes, W. van der Hoek, and M. Wooldridge. Reasoning about coalitional games. *Artif. Intell.*, 173(1):45–79, 2009.

[23] S. Schewe. ATL* Satisfiability is 2ExpTime-Complete. In *ICALP'08*, LNCS 5126, pages 373–385. Springer, 2008.

[24] Yoav Shoham and Moshe Tennenholtz. On the synthesis of useful social laws for artificial agent societies. In *AAAI'92*, pages 276–281. AAAI Press, 1992.

[25] Wiebe van der Hoek, Mark Roberts, and Michael Wooldridge. Social laws in alternating time: effectiveness, feasibility, and synthesis. *Synthese*, 156:1–19, 2007.

Runtime Enforcement of Timed Properties

Srinivas Pinisetty
INRIA Rennes, France
First.Last@inria.fr

Yliès Falcone
LIG, Université Grenoble I, France
First.Last@ujf-grenoble.fr

Thierry Jéron
INRIA Rennes, France
First.Last@inria.fr

Hervé Marchand
INRIA Rennes, France
First.Last@inria.fr

Antoine Rollet
LaBRI, Université de Bordeaux - CNRS
First.Last@labri.fr

Omer Nguena Timo
LaBRI, Université de Bordeaux - CNRS
nguena@labri.fr

Abstract

Runtime enforcement is a powerful technique to ensure that a running system respects some desired properties. Using an enforcement monitor, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies to a property. Runtime enforcement has been extensively studied over the last decade in the context of untimed properties.

This work introduces runtime enforcement of timed properties. We revisit the foundations of runtime enforcement when time between events matters. We show how runtime enforcers can be synthesized for any safety or co-safety timed property. Proposed runtime enforcers are time retardant: to produce an output sequence, additional delays are introduced between the events of the input sequence to correct it. Runtime enforcers have been prototyped and our simulation experiments validate their effectiveness.

1 Introduction

Runtime verification [1, 2, 3, 4, 5, 6] (resp. enforcement [7, 8, 9]) refers to the theories, techniques, and tools aiming at checking (resp. ensuring) the conformance of the executions of systems under scrutiny w.r.t. some desired property. A central concept is the verification or enforcement *monitor* that is generally synthesized from the property expressed in a high-level formalism. When the monitor is only dedicated to verification, it is a decision procedure emitting verdicts stating the correctness of the (partial) observed trace generated from the system execution. In runtime enforcement, an enforcement monitor (EM) is used to transform some (possibly) incorrect execution sequence into a correct sequence w.r.t. the property of interest. Some of the results on how untimed properties can be enforced at runtime are [7, 8, 9]. Regarding timed properties, most of the results sofar are related to how to verify them at runtime [10, 12, 13], but not about how to enforce them. More details on related work are provided in [15]. We focus on *online enforcement of timed properties*. To the best of our knowledge, no approach was proposed to enforce timed properties. Motivations for extending runtime enforcement to timed properties abound. First, timed properties are a more precise tool to specify desired behaviors of systems since they allow to explicitly state how time should elapse between two events. Moreover, several applications of runtime enforcement of timed properties can be considered. For instance, in the context of security monitoring, enforcement monitors can be used as firewalls to prevent denial of service attacks by ensuring a minimal delay between input events (carrying some request for a protected server). On a network, enforcement monitors can be used to synchronize streams of events together, or, ensuring that a stream of events conforms to the pre-conditions of some service.

Contributions. We propose a context where, under some reasonable assumptions, runtime enforcement of timed properties is possible. Runtime enforcement monitors are built from safety and co-safety properties expressed by timed automata. In contrast with previous runtime enforcement approaches, we afford only the primitives of being able to delay the input events to our enforcer. By possibly increasing delays between events of the input sequence, the output timed sequence conforms to the property.

Delays are modified by monitors using an internal memory where (sequence of) events are stored and released after appropriate delays. Experiments have been performed on prototype monitors to show their effectiveness and the feasibility of our approach.

2 Preliminaries and Notation

Untimed notions. An alphabet is a finite set of elements. A (finite) word over an alphabet A is a finite sequence of elements of A. The *length* of a word w is noted $|w|$. The empty word over A is denoted by ε_A or ε when clear from context.The set of all (resp. non-empty) words over A is denoted by A^* (resp. A^+). A *language* over A is a subset $\mathscr{L} \subseteq A^*$. The concatenation of two words w and w' is noted $w \cdot w'$. A word w' is a prefix of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$. For a word w and $1 \leq i \leq |w|$, the i-th letter (resp. prefix of length i, suffix starting at position i) of w is noted $w(i)$ (resp. $w_{[\cdots i]}$, $w_{[i\cdots]}$) – with the convention $w_{[\cdots 0]} \stackrel{\text{def}}{=} \varepsilon$. $\text{pref}(w)$ denotes the set of prefixes of w and by extension, $\text{pref}(\mathscr{L}) \stackrel{\text{def}}{=} \{\text{pref}(w) \mid w \in \mathscr{L}\}$ the prefix of $\mathscr{L}$. $\mathscr{L}$ is said to be *prefix-closed* whenever $\text{pref}(\mathscr{L}) = \mathscr{L}$ and *extension-closed* whenever $\mathscr{L} = \mathscr{L} \cdot A^*$. Given a tuple of symbols $e = (e_1, \ldots, e_n)$, $\Pi_i(e)$ is the projection of e on its i^{th} element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$).

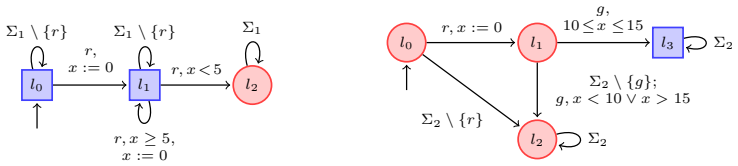
Timed languages. Let $\mathbb{R}_{\geq 0}$ denote the set of non negative real numbers, and Σ a finite alphabet of *actions*. A pair $(\delta, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$ is called an *event*. We note $\text{del}(\delta, a) = \delta$ and $\text{act}(\delta, a) = a$ the projections of events on delays and actions, respectively. A *timed word* over Σ is a finite sequence of events ranging over $(\mathbb{R}_{\geq 0} \times \Sigma)^*$. For $\sigma = (\delta_1, a_1) \cdot (\delta_2, a_2) \cdots (\delta_n, a_n)$, δ_i $(2 \leq i \leq n)$ is the delay between a_{i-1} and a_i and δ_1 the time elapsed before the first action. Note that the alphabet is infinite in this case. Nevertheless, notions and notations defined above (related to length, concatenation, prefix, etc) naturally extend to timed words. *The sum of delays* of a timed word σ is noted $\text{time}(\sigma)$. Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we define the *observation of σ at time t* as the timed word $\text{obs}(\sigma, t) \stackrel{\text{def}}{=} \max\{\sigma' \mid \sigma' \preccurlyeq \sigma \wedge \text{time}(\sigma') \leq t\}$, i.e., the longest prefix of σ with a sum of delays less than t. The *untimed projection* of σ is $\Pi_\Sigma(\sigma) \stackrel{\text{def}}{=} a_1 \cdot a_2 \cdots a_n$ in Σ^* (i.e., delays are ignored). A *timed language* is any subset $\mathscr{L} \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. We define the following order on timed words: σ' *delays* σ (noted $\sigma' \preccurlyeq_d \sigma$) if $\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$ and $\forall i \leq |\sigma'|, \text{del}(\sigma(i)) \leq \text{del}(\sigma'(i))$.

Timed Automata. Let $X = \{X_1, \ldots, X_k\}$ be a finite set of *clocks*. A *clock valuation* for X is a function ν from X to $\mathbb{R}^X_{\geq 0}$ where $\mathbb{R}^X_{\geq 0}$ denotes the valuations of X. For $\nu \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ is the valuation assigning $\nu(X_i) + \delta$ to each clock X_i of X. Given a set of clocks $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation ν where all clocks in X' are assigned to 0. $\mathscr{G}(X)$ denotes the set of clock constraints defined as boolean combinations of simple constraints of the form $X_i \bowtie c$ with $X_i \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathscr{G}(X)$ and $\nu \in \mathbb{R}^X_{\geq 0}$, we write $\nu \models g$ when $g(\nu) \equiv true$.

Definition 1 (Timed automaton)**.** A *timed automaton* (TA) is a tuple $\mathscr{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$, s.t. L is a finite set of *locations* with $l_0 \in L$ the *initial location*, X is a finite set of clocks, Σ is a finite set of *events*, $\Delta \subseteq L \times \mathscr{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*, and $G \subseteq L$ is a set of accepting locations.

The *semantics* of a TA is a timed transition system $[\![\mathscr{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$ where $Q = L \times \mathbb{R}^X_{\geq 0}$ is the (infinite) set of *states*, $q_0 = (l_0, \nu_0)$ is the initial state where ν_0 is the valuation that maps every clock to 0, $F_G = G \times \mathbb{R}^X_{\geq 0}$ is the set of of accepting states, $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ is the set of transition *labels*, i.e., pairs composed of a delay and an action. The transition relation $\rightarrow \subseteq Q \times \Gamma \times Q$ is a set of transitions of the form $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ with $\nu' = (\nu + \delta)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ s.t. $\nu + \delta \models g$ for $\delta \geq 0$.

In the following, we consider a timed automaton $\mathscr{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ with its semantics $[\![\mathscr{A}]\!]$. $\mathscr{A}$ is *deterministic* whenever for any (l, g_1, a, Y_1, l'_1) and (l, g_2, a, Y_2, l'_2) in Δ, $g_1 \wedge g_2$ is *false*. $\mathscr{A}$ is *complete* whenever for any location $l \in L$ and every event $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labeled by a is *true*. In the remainder of this paper, we shall consider only deterministic timed automata.



(a) A safety TA for φ_1 (b) A co-safety TA for φ_2

Figure 1: Example of Timed Properties

A *run* ρ from $q \in Q$ is a sequence of moves in $[\![\mathscr{A}]\!]$ of the form: $\rho = q_0 \xrightarrow{(\delta_1,a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n,a_n)} q_n$. The set of runs from $q_0 \in Q$ is denoted $Run(\mathscr{A})$ and $Run_{F_G}(\mathscr{A})$ denotes the subset of runs *accepted* by $\mathscr{A}$, i.e., ending in F_G. The *trace* of a run ρ is the timed word $(\delta_1,a_1)\cdot(\delta_2,a_2)\cdots(\delta_n,a_n)$. We note $\mathscr{L}(\mathscr{A})$ the set of traces of $Run(\mathscr{A})$. We extend this notation to $\mathscr{L}_{F_G}(\mathscr{A})$ in a natural way.

Timed Properties A timed property is defined by a timed language $\varphi \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Given a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$. In the sequel, we shall be interested in safety and co-safety timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). Safety (resp. co-safety) properties can be characterized by prefix-closed (resp. extension-closed) languages. We consider only the sets of safety and co-safety properties that can be represented by timed automata (Definition 1).

Definition 2 (Safety and Co-safety TA)**.** A complete and deterministic TA $\langle L, l_0, X, \Sigma, \Delta, G\rangle$, where $G \subseteq L$ is the set of accepting locations, is said to be:

- a *safety* TA if $l_0 \in G \wedge \nexists \langle l,g,a,Y,l'\rangle \in \Delta, l \in L \setminus G \wedge l' \in G$;
- a *co-safety* TA if $l_0 \notin G \wedge \nexists \langle l,g,a,Y,l'\rangle \in \Delta, l \in G \wedge l' \in L \setminus G$.

It is easy to check that safety and co-safety TAs define safety and co-safety properties.

Example 3 (Safety and co-safety TA)**.** Fig. 1a and 1b present two properties formalized with safety and co-safety TA. Accepting locations are represented by squares. The safety TA formalizes the property φ_1 defined over $\Sigma_1 = \{a, r\}$: *"There should be a delay of at least 5 time units between any two user requests (r)"*. The co-safety TA formalizes the property φ_2 defined over $\Sigma_2 = \{r, g, a\}$: *"The user can perform an action a only after a successful authentication, i.e., after sending a request r and receiving a grant g. After an r, g should occur between 10 and 15 time units"*.

3 Enforcement Monitoring in a Timed Context

Roughly speaking, both in the timed and untimed settings, the purpose of an enforcement monitor (EM) is to read some (possibly incorrect) input sequence σ produced by a running system (input to the enforcer), and to transform it into an output sequence o that is correct w.r.t. a property φ, here modeled by a TA.

Definition 4. For a given property φ, an *enforcement function* is a function E from $(\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0}$ to $(\mathbb{R}_{\geq 0} \times \Sigma)^*$.

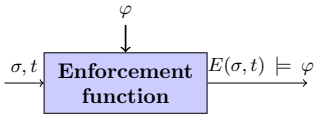


Figure 2: Enforcement function E

An enforcement function E transforms some timed word σ given as input and possibly incorrect w.r.t. the desired property (see Fig. 2). The resulting output $E(\sigma,t)$ at time t is a timed word with same actions, but possibly increased delays between actions so that it satisfies the property. Similar to the untimed setting, additional constraints on $E(\sigma,t)$, namely *soundness* and *transparency*, are required on actions. However, in the timed setting, those constraints also depend on both delays between events and the class of the enforced property, as we shall discuss later.

An enforcement function E is realized by an *enforcement monitor EM*. This monitor is equipped with a memory and a set of enforcement operations used to store and dump some timed events to and

from the memory, respectively. The memory of an *EM* is basically a queue containing a timed word, the received actions with increased delays that have not been released yet.

In the following sections, we will present enforcement monitors for both safety and co-safety properties and analyze constraints on the associated enforcement functions.

4 Enforcement of Safety Properties

In this section we focus on the enforcement of a safety property φ specified by a safety automaton $\mathscr{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ and its associated semantics $[\![\mathscr{A}]\!] = \langle Q, q_0, \Gamma, \to, F_G\rangle$. Without loss of generality, we assume that the set of locations $L \setminus G$ is reduced to a singleton $\{Bad\}$. Given φ, and a timed word σ, an enforcement function E for φ should satisfy the following soundness, transparency and optimality conditions.

Definition 5 (Soundness, transparency and optimality)**.** Let $E : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ be an enforcement function for a safety property φ. E is:

- *sound* if $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma,t) \models \varphi$;
- *transparent* if $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma,t) \preccurlyeq_d \text{obs}(\sigma,t) \wedge \text{time}(E(\sigma,t)) \leq t$.

If E is both sound and transparent, we say that it is *optimal* if, for any input $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, at any time $t \in \mathbb{R}_{\geq 0}$, the following constraints hold:

(Op1) $\nexists \omega', \omega' \models \varphi \ \wedge\ \omega' \preccurlyeq_d obs(\sigma,t) \ \wedge\ |\omega'| > |E(\sigma,t)|$

(Op2) $\forall i \in [1, |E(\sigma,t)|], \neg\nexists \delta'' \in \mathbb{R}_{\geq 0}, \quad \text{del}(\text{obs}(\sigma,t)(i)) \leq \delta'' \leq \text{del}(E(\sigma,t)(i)) \wedge E(\sigma,t)_{[\cdots i-1]} \cdot (\delta'', \text{act}(E(\sigma,t)(i))) \models \varphi$

Soundness means that, at any time t, the produced timed word should satisfy the property φ. Transparency means that, at any time instant t, the output $E(\sigma,t)$ delays the input $\text{obs}(\sigma,t)$: the enforcement function should not modify the order of events, should not reduce the delays between consecutive events, and should not produce outputs faster than inputs. Optimality means that the enforcement function should provide the output as soon as possible. The optimality condition **(Op1)** extends the requirement on the output sequences of the enforcement function in the untimed case (cf. [9]): at any time instant t, the output sequence $E(\sigma,t)$ should be the longest correct timed word delaying the input sequence $\text{obs}(\sigma,t)$. Here, taking physical time into account, **(Op2)** requires that the input and output sequences are as close as possible w.r.t. physical observation, i.e., every prefix of $E(\sigma,t)$ has the shortest possible last delay.

We now design an enforcement monitor whose semantics effectively realizes the enforcement function as described Definition 5.

Definition 6 (Enforcement Monitor for safety)**.** An enforcement monitor for φ is a transition system $EM = \langle C, C_0, \Gamma_{EM}, \hookrightarrow\rangle$ s.t.:

- $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B} \times Q$ is the set of configurations;
- the initial configuration is $C_0 = \langle \varepsilon, 0, 0, \texttt{tt}, q_0\rangle \in C$;
- $\Gamma_{EM} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\varepsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\varepsilon\}\big)$ is the input-operation-output alphabet, where $Op = \{\text{store}(\cdot), \text{dump}(\cdot), \text{del}(\cdot)\}$;
- $\hookrightarrow \subseteq C \times \Gamma_{EM} \times C$ is the transition relation defined as the smallest relation obtained by the following rules applied in the following order:
  - store: $\langle \sigma_s, \delta, d, \texttt{tt}, q\rangle \overset{(\delta,a)/\text{store}(\delta',a)/\varepsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta', a), 0, d, (\delta' \neq \infty), q'\rangle$ with:
    - $\delta' = \text{update}_s(q, a, \delta)$, where $\text{update}_s$is the function defined as:

$$\begin{array}{rcl} Q \times \Sigma \times \mathbb{R}_{\geq 0} & \to & \mathbb{R}_{\geq 0} \\ (q,a,\delta) & \mapsto & \begin{cases} \infty & \text{if } \forall \delta' \in \mathbb{R}_{\geq 0}, \forall q_1 \in Q, (\delta' \geq \delta \wedge q \overset{(\delta',a)}{\to} q_1) \Rightarrow q_1 \notin F_G \\ \min\{\delta' \in \mathbb{R}_{\geq 0} \mid \exists q_1 \in F_G, q \overset{(\delta',a)}{\to} q_1 \wedge \delta' \geq \delta\} \end{cases} \end{array}$$

    - q' is defined as $q \overset{(\delta',a)}{\to} q'$ if $\delta' < \infty$ and $q' = q$ otherwise;

– dump: $\langle(\delta,a)\cdot\sigma_s,s,\delta,b,q\rangle \overset{\varepsilon/\text{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle\sigma_s,s,0,b,q\rangle$ if $\delta\neq\infty$;

– delay: $\langle\sigma_s,s,d,b,q\rangle \overset{\varepsilon/\text{del}(\delta)/\varepsilon}{\hookrightarrow} \langle\sigma_s,s+\delta,d+\delta,b,q\rangle$.

A configuration $\langle\sigma_s,s,d,b,q\rangle$ of the *EM* consists of the current stored sequence (i.e., the memory content) σ_s, two clock values s and d indicating respectively the time elapsed since the last store and dump operations, a Boolean b indicating whether the underlying enforced property is satisfied or not on the output sequence, and q the current state of $[\![\mathscr{A}]\!]$ reached after processing the sequence already released followed by the timed word in memory. Regarding its alphabet, in the input (resp. output) sequence, the *EM* either lets time elapse and no event is read or released, or reads and stores (resp. dumps and releases) a symbol event after some delays. Semantics rules can be understood as follows:

- The *store* rule is executed upon the reception of an event (δ,a). The timed event (δ',a) is appended to the memory content, where δ' is the minimal delay that has to be waited so that the property remains satisfied – if such a delay exists. The value of s is then reinitialized to 0. If a delay can be found through the update_s function, q is updated to the state that will be reached by appending the timed event (δ',a) to the output sequence concatenated with the contents of the memory, and b remains `tt` and becomes `ff` otherwise.
- The *dump* rule is executed when the value of d is equal to the delay of the first timed event in the memory. The value of d is then reinitialized to 0. The first event in memory is suppressed (and released from the enforcer). Other elements of the configuration remain unchanged.
- The *delay* rule adds the time elapsed δ to the current values of s and d when no store nor dump operation is possible.

We define the language of runs of an enforcement monitor *EM*:

$$\mathscr{L}(EM)\subseteq(\Gamma_{EM})^* = \Big(\big((\mathbb{R}_{\geq 0}\times\Sigma)\cup\{\varepsilon\}\big)\times Op\times\big((\mathbb{R}_{\geq 0}\times\Sigma)\cup\{\varepsilon\}\big)\Big)^*$$

It is worth noticing that enforcement monitors are deterministic. Hence, given $\sigma\in(\mathbb{R}_{\geq 0}\times\Sigma)^*$ and $t\in\mathbb{R}_{\geq 0}$, let $w\in\mathscr{L}(EM)$ be the unique maximal sequence such that $\Pi_\varepsilon\Big(\bigodot_{i\in[1,|w|]}(\Pi_1(w(i)))\Big)=\text{obs}(\sigma,t)$, where Π_ε is the projection that erases ε from words in $((\mathbb{R}_{\geq 0}\times\Sigma)\cup\{\varepsilon\})^*$. Now, we define the enforcement function E associated to EM as

$$\forall\sigma\in(\mathbb{R}_{\geq 0}\times\Sigma)^*,\ \forall t\in\mathbb{R}_{\geq 0},\ E(\sigma,t)=\Pi_\varepsilon\Big(\bigodot_{i\in[1,|w|]}(\Pi_3(w(i)))\Big) \tag{1}$$

Proposition 7. *Given an enforcement monitor EM for a safety property φ and E defined as in Eq. (1), E verifies the soundness, transparency and optimality conditions of Definition 5.*

5 Enforcement of Co-safety Properties

Let us now focus on the enforcement of co-safety properties. An *EM* for a co-safety property, starts to release events only after observing the minimal sequence of input events which can lead to a good location. The sum of the delays of this minimal sequence is optimized rather than each delay as was the case for a safety property. An *EM* for a co-safety property is defined as a transition system similar to an *EM* for as safety property with some minor adaptations. Definitions and detailed explanation with examples are presented in [15].

6 Implementation and Experiments

The implementation of an enforcement monitor (EM) consists of two processes running concurrently (Store and Dump) and a memory. The Store process models the store rules. The memory contains the timed words σ_s. The Dump process reads events stored in the memory and releases them as output after the required amount of time. The algorithms are presented and described in detail in [15].

Enforcement monitors for safety and co-safety properties, have been implemented in prototype tool of 500 LOC using Python. The tool also uses UPPAAL [11] as a library to implement the update function and the pyuppaal library to parse UPPAAL models written in XML. More details about the experiments and performance evaluation using a simulated system are described in [15].

7 Conclusion and Future Work

This work introduces runtime enforcement for timed properties and provides a complete framework. We consider safety and co-safety properties described by timed automata. We propose adapted notions of enforcement monitors with the possibility to delay some input actions in order to satisfy the required property. For this purpose, the enforcement monitor can store some actions for a certain time period. We propose a set of enforcement rules ensuring that outputs not only satisfy the required property (if possible), but also with the "best" delay according to the current situation. We describe how to realize the enforcement monitor using concurrent processes, how it has been prototyped and experimented.

We introduced the first steps to runtime enforcement of (continuous) timed properties. However, several research questions remain open. As this approach targets explicitly safety and co-safety properties, it seems desirable to investigate whether more expressive properties can be enforced. We expect to extend our approach to Boolean combinations of timed safety and co-safety properties, and more general properties. This requires further investigation since the update function would have to be adapted. A precise characterization of *enforceable timed properties* would thus be possible, as was the case in the untimed setting [4, 14]. A more practical research perspective is to study the implementability of the proposed approach, e.g., using *robustness* of timed automata.

References

[1] Thati, P., Rosu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theor. Comput. Sci. **113** (2005) 145–162

[2] Chen, F., Rosu, G.: Parametric trace slicing and monitoring. In Kowalewski, S., Philippou, A., eds.: TACAS. Volume 5505 of LNCS., Springer (2009) 246–261

[3] Nickovic, D., Piterman, N.: From MTL to deterministic timed automata. In Chatterjee, K., Henzinger, T.A., eds.: FORMATS. Volume 6246 of LNCS., Springer (2010) 152–167

[4] Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20** (2011) 14

[5] Basin, D.A., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In Khurshid, S., Sen, K., eds.: RV. Volume 7186 of LNCS., Springer (2011) 260–275

[6] Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: FM 2012: 18th International symposium on Formal Methods. (2012) Accepted for publication. To appear.

[7] Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000)

[8] Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transaction Information System Security. **12** (2009)

[9] Falcone, Y.: You should better enforce than verify. In: Runtime Verification. (2010) 89–105

[10] Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Formal Modeling and Analysis of Timed Systems. (2007) 304–319

[11] Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) **1** (1997) 134–152

[12] Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In Asarin, E., Bouyer, P., eds.: FORMATS. Volume 4202 of LNCS., Springer (2006) 274–289

[13] Colombo, C., Pace, G.J., Schneider, G.: Safe runtime verification of real-time properties. In: Formal Modeling and Analysis of Timed Systems, 7th International Conference (FORMATS). Volume 5813 of LNCS., Budapest, Hungary (2009) 103–117

[14] Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? STTT **14** (2012) 349–382

[15] Pinisetty, S. and Falcone, Y. and Jéron, T. and Marchand, H. and Rollet, A. and Nguena Timo, O.: Runtime Enforcement of Timed Properties. In proceedings of the Third International Conference on Runtime Verification RV 2012

Construction and Verification of Unfoldings for Petri Nets with Read Arcs

César Rodríguez*
LSV (ENS Cachan & CNRS), France
cesar.rodriguez@lsv.ens-cachan.fr

Abstract

A contextual net is a Petri net extended with read arcs, which allow transitions to check for tokens without consuming them. Contextual nets allow for better modelling of concurrent read access than Petri nets, and their unfoldings can be exponentially more compact than those of a corresponding Petri net. In this talk, we overview the theoretical results that lead to the implementation of the first contextual net unfolder and reachability checker. We provide experimental evidence supporting the thesis that verification based on contextual unfoldings is at least as efficient as, and often more than, verification based on ordinary unfoldings.

Unfoldings of Petri nets. Petri nets are a means for reasoning about concurrent, distributed systems. They explicitly express notions such as concurrency, causality, and independence. The *unfolding* of a bounded Petri net N is a safe and acyclic Petri net $\mathscr{U}_N$ equipped with a *folding morphism* $f\colon \mathscr{U}_N \to N$ that maps transitions and places of $\mathscr{U}_N$, respectively called *events* and *conditions*, to transitions and places of N, respectively. The unfolding satisfies three important properties:

1. The sequence $f(\sigma)$ labelling any run σ of $\mathscr{U}_N$ is a run of N.
2. For any marking $\widehat{m}$ reachable in $\mathscr{U}_N$, the labelling $f(\widehat{m})$ is a reachable marking of N.
3. For any marking m reachable in N, there is some reachable marking $\widehat{m}$ of $\mathscr{U}_N$ with $f(\widehat{m}) = m$.

We actually call *complete* (w.r.t. N) any net satisfying (3). Every reachable marking of $\mathscr{U}_N$ *represents*, thus, a reachable marking of N, and all markings reachable in N are represented in $\mathscr{U}_N$. In Fig. 1, the unfolding of the Petri net (b) is shown in (c).

In general $\mathscr{U}_N$ is infinite. Ken McMillan was the first to note [11] that a *finite* complete *prefix* $\mathscr{P}_N$ of $\mathscr{U}_N$ could be constructed by appropriately *cutting off* infinite branches of $\mathscr{U}_N$, provided that N has finitely many states. He also realized that checking coverability or deadlock-freeness of N is PSPACE-complete in N but only NP-complete in $\mathscr{P}_N$, due to its acyclic structure. Of course $\mathscr{P}_N$ can be exponentially larger than N, but this is often not the case for highly concurrent systems.

The publication of [11] triggered a large body of research. To name a few items, the necessary size of $\mathscr{P}_N$ has been reduced [6], efficient tools have been implemented [15, 10], and unfolding-based verification methods have been developed [4, 7, 9]. See [5] for a survey.

*This is joint work with Stefan Schwoon and Paolo Baldan.

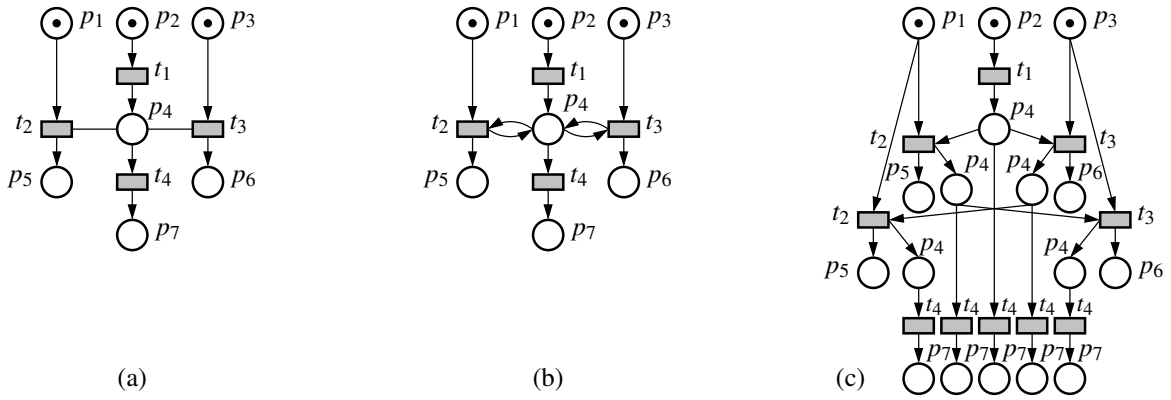


Figure 1: (a) a c-net; (b) its encoding into a Petri net; (c) unfolding of (b)

Contextual nets and contextual unfoldings. *Contextual nets* (c-nets) are Petri nets extended with *read arcs*, i.e., arcs allowing a transition to check for the presence of tokens without consuming them. The *context* $\underline{t}$ of a transition t is the set of places linked with a read arc to t. The context of a place is defined analogously. Fig. 1 (a) shows a c-net with two read arcs connected to p_4, where we have $\underline{t_2} = \underline{t_3} = \{p4\}$ and $\underline{p_4} = \{t_2, t_3\}$. As ordinary Petri nets, c-nets can be unfolded into the so-called *contextual unfoldings* [16, 3], acyclic and 1-safe *c-nets*. Infinite branches of the contextual unfolding can also be pruned with a suitable algorithm, yielding a *complete contextual prefix*. Deciding reachability of a c-net using a complete contextual prefixes is also NP-complete.

While contextual unfoldings have the advantages of ordinary unfoldings, they enjoy additional ones. For every c-net N, one can build an equivalent Petri net N' obtained by substituting read arcs in N for consume-produce loops. In Fig. 1, (b) has been obtained in this way from (a). In interleaving semantics, N and N' have essentially the same firing sequences and reachable markings. Every c-net can thus be encoded into, and verified as, a Petri net.

Crucially, the complete contextual unfolding prefix $\mathscr{P}_N$ of N can be *up to exponentially more compact* than $\mathscr{P}_{N'}$. A place read by multiple transitions, e.g. p_4 in Fig. 1 (a), yields in N' a choice between those transitions, which causes all their interleavings to be *explicitly* present in $\mathscr{P}_{N'}$ – unfoldings fail to treat choices more intelligently. Such transitions are, however, treated as concurrent in $\mathscr{P}_N$, and therefore compactly represented. Concurrent read access to common resources happens naturally in several applications, such as concurrent database access, concurrent constraint programs, or asynchronous circuits. Other encodings from c-nets into Petri nets are possible, but they lead to similar problems [1].

The referred explosion is shown in Fig. 1. The unfolding of the (already acyclic) c-net (a) is a c-net isomorphic to (a), with the isomorphism being the folding morphism f. In (c), the unfolding operation has explicitly interleaved all occurrences of the *reading* transitions t_2 and t_3, that are in conflict in (b) – but concurrent in (a). This led to 5 occurrences of t_4. If (a) was generalized to n readers, the contextual unfolding would still be isomorphic to the c-net, but the unfolding of its Petri-net encoding would at least have $n!$ occurrences of t_4. Verification techniques based on contextual unfoldings can, thus, profit of their additional conciseness and will lead, we believe, to better verification tools.

In recent work [2], Baldan et al. established theoretical foundations for constructing c-net unfoldings. A fundamental phenomenon present in c-nets but not in Petri nets is *asymmetric conflict* between an event reading a token and an event consuming it, for instance t_2 and t_4 in Fig. 1 (a). This phenomenon forces the c-net unfolding construction to keep track of the so-called *histories*. If e is an event, then a *history* of

e is any set H of events present in a run of $\mathscr{U}_N$ that contains e and such that any run that fires exactly all events in H, fires e last. In the unfolding of Fig. 1 (a), the event labelled by t_4 has four histories (t_2 and t_3 may or may not be included).

Traditionally, the unfolding algorithm for Petri nets incrementally constructs a complete prefix starting from the initial conditions and extending it with one event at a time. Computing such *extensions* requires solving the coverability problem for the conditions enabling the event on the currently constructed part of the prefix. To achieve this, Petri net unfolders often rely on a *concurrency relation*, which stores coverability information in an implicit fashion. In order to keep the prefix finite, they additionally mark certain events as *cut-offs*: the pruning points of the otherwise potentially infinite branches.

The need to deal with histories during c-net unfolding construction impacts all aspects of the unfolding algorithm. Every event in the prefix is annotated with a set of its histories. Possible extensions are not anymore events, but *enriched events*, pairs (e,H) where e is an event and H is a history of e. Computing possible extensions requires now solving a variant of the coverability problem where histories are involved. The notion of cut-off is also lifted to enriched events.

Results and current work. Building on [2], we proposed in [14] algorithms and data structures to efficiently cope with the additional complexity introduced by histories. Specifically, we proposed algorithms to compute the extensions of a prefix, to decide which events are cutoffs, and data structures to represent histories.

Our algorithm stores every history in a node-labelled directed graph called the *history graph*. Histories have a compositional structure. Every history H of an event e is a union of $\{e\}$ and the histories of events in $^\bullet(\underline{e})$, $^{\bullet\bullet}e$, and $\underline{(^\bullet e)}$. Using this fact, H is implicitly represented in the history graph as a node labelled with e, where outgoing edges point to the histories in the decomposition. Typical operations on histories conveniently reduce to operations on the node and its neighborhood.

We also introduced the notion of *enriched condition*, pairs (c,H) where c is a condition of the unfolding and H is either a history of any event in $^\bullet c$, or a union of histories of events in $\underline{c}$. Enriched conditions are at the heart of the algorithm for computing possible extensions, as we sketch now.

A *concurrency relation* on enriched conditions was presented as a means to compute the possible extensions of the prefix. Intuitively, this relation contains pairs of enriched conditions (c,H), (c',H') such that c and c' are marked when one fires *together* their histories, i.e., the set $H \cup H'$ is a configuration that marks c,c'. Deciding if the prefix has a possible extension of the form (e,H), where $f(e) = t$, amounts to searching for sets of conditions D, C where $f(D) = {}^\bullet t$, $f(C) = \underline{t}$, and there exists one history for every condition in $D \cup C$ such that the associated enriched conditions are concurrent.

Including new events and conditions in the prefix gives rise to new enriched conditions which may be concurrent to existing enriched conditions. The concurrency relation needs, thus, to be updated after every extension. To achieve this, we presented an *inductive characterization* of the concurrency relation, which identifies an efficient algorithm for computing these udpates. The algorithm profits from the concurrency information already present in the relation to compute the concurrency of new enriched conditions.

The overhead imposed by histories questioned the practical interest of c-net unfoldings: while the final unfolding could be quite compact, the intermediate product of the construction could be exponentially larger than the result. We implemented the method in the tool CUNF [12]. We aimed at building a competent c-net unfolder that could be compared to existing tools, ensuring that the theoretical gains were put into practice. Experimental comparisons over a standard benchmark, thus containing nets not specially targeted towards contextual unfolding, suggested that c-net unfolding is often faster and yields

smaller unfoldings [14, 1].

Once the practical benefits of c-net unfoldings were established, we focused on verification methods based on them. In [13], we presented a method for checking deadlock-freenness or coverability. Here, these problems were reduced to the satisfiability of a propositional formula generated from the unfolding. We also proposed a number of optimizations reducing the encoding size. A tool implementing this method is distributed together with CUNF. In our benchmarks, the accumulated SAT solving time of this tool beats that of previously existing verification tools.

Currently we work on the definition and construction of *contextual merged processes*. A merged process is a condensed representation of the state space of a Petri net, which copes with certain sources of state space explosion in net unfoldings, like sequences of choices or non-safeness [8]. Contextual merged processes promise to yield an exponential gain in compactness over ordinary merged processes for certain classes of nets.

Plan of the talk. In this talk, we present results about the construction and verification of unfoldings for c-nets. After recalling the interest of c-net unfoldings in verification, we present an efficient method for constructing contextual unfoldings [14], and report on experiments performed with CUNF. We provide experimental evidence supporting the thesis that building contextual unfoldings is at least as efficient as, and often more than, building ordinary unfoldings.

We next focus on the analysis of c-nets by means of their unfoldings. We present a reduction of the deadlock and coverability problems into SAT [13]. Time permitting, we then compare, over a series of experiments, the performance of our method and other unfolding-based verification tools.

All results presented in this talk have been published in [14], [13], and [1].

References

[1] Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König, César Rodríguez, and Stefan Schwoon. Efficient Unfolding of Contextual Petri Nets. *Theoretical Computer Science*, 449:2 – 22, 2012.

[2] Paolo Baldan, Andrea Corradini, Barbara König, and Stefan Schwoon. McMillan's complete prefix for contextual nets. *ToPNoC*, 1:199–220, 2008. LNCS 5100.

[3] Paolo Baldan, Andrea Corradini, and Ugo Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. In *Proc. FoSSaCS*, volume 1378 of *LNCS*, pages 63–80, 1998.

[4] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In *Proc. SPIN*, volume 2057 of *LNCS*, pages 37–56, 2001.

[5] Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[6] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

[7] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fund. Inf.*, 37(3):247–268, 1999.

[8] Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. Merged processes – a new condensed representation of Petri net behaviour. *Act. Inf.*, 43(5):307–330, 2006.

[9] Victor Khomenko and Maciej Koutny. LP deadlock checking using partial order dependencies. In *Proc. CONCUR*, LNCS 1877, pages 410–425, 2000.

[10] Viktor Khomenko. PUNF. `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/`.

[11] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV*, LNCS 663, pages 164–177, 1992.

[12] César Rodríguez. CUNF. `http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/`.

[13] César Rodríguez and Stefan Schwoon. Verification of Petri Nets with Read Arcs. In *Proc. of CONCUR'12*, volume 7454 of *LNCS*, September 2012.

[14] César Rodríguez, Stefan Schwoon, and Paolo Baldan. Efficient contextual unfolding. In *Proc. of CONCUR'11*, volume 6901 of *LNCS*, pages 342–357, September 2011.

[15] Stefan Schwoon. MOLE. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`.

[16] Walter Vogler, Alexei L. Semenov, and Alexandre Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proc. of CONCUR'98*, volume 1466 of *LNCS*, pages 501–516, 1998.

On Minimality and Equivalence of Petri Nets (Extended Abstract)

Annegret K. Wagler, Jan-Thierry Wegener*
Université Blaise Pascal (Clermont-Ferrand II)
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes
BP 10125, 63173 Aubière Cedex, France

Annegret.WAGLER@univ-bpclermont.fr
wegener@isima.fr

Abstract

The context of this work is the reconstruction of Petri net models from experimental data. Our approach aims at generating all possible models which explain a given set of experimental data. This typically leads to big solution sets where, however, many solutions are similar or redundant. In order to keep the solution set as small as possible, while guaranteeing its completeness, the idea is to generate only "minimal" Petri nets in the sense that all other networks fitting the data contain the reconstructed ones.

In this extended abstract we consider Petri nets with three types of extensions: capacities on places, priorities on transitions and control-arcs. We define both an equivalence relation for Petri nets and an inclusion relation for equivalent Petri nets, taking all above extensions into account. Finally, we discuss the impact of our results to our approach of reconstructing Petri net models.

1 Introduction

Understanding different phenomena is one of the aims of system biology; e.g., observing responses of cells to environmental changes, host-pathogen interactions, and effects of gene defects. Several different types of mathematical models have been developed to model and explain experimental data, see for instance [8, 11].

Here we focus on Petri nets, a framework which turned out to coherently model both static interactions in terms of networks and the dynamic processes in terms of state changes [1, 2, 9, 13, 15]. In a Petri net $\mathscr{P} = (P, T, \mathscr{A}, w)$ the set of places P reflects the involved components and the set of transitions T reflects the interactions between the different components, linked to the places by directed arcs in $\mathscr{A}$, equipped with arc weights $w : \mathscr{A} \to \mathbb{N}$. System states are represented with tokens on the places, i.e., a system state is a vector $\mathbf{x} \in \mathbb{N}^{|P|}$. In this work, we consider capacitated Petri nets $(\mathscr{P}, \text{cap})$, i.e., a Petri net $\mathscr{P}$ equipped with a capacity function $\text{cap} : P \to \mathbb{N}$ which reduces the potential state space to $\mathscr{X} = \{\mathbf{x} \in \mathbb{N}^{|P|} \mid x_p \leq \text{cap}(p)\}$. The dynamic processes are represented by sequences of state changes, performed by switching or firing enabled transitions (see Section 2).

To obtain models from experimentally observed sequences of state changes, an exact, exclusively data-driven, combinatorial approach was developed in [5, 6, 12, 17]. This approach takes as input a set P of places and discrete time-series data $\mathscr{X}'$ given by sequences $(\mathbf{x}^1, \ldots, \mathbf{x}^m)$ of experimentally observed system states. The first state x^1 in such a sequence is called initial state. A capacitated Petri net

*This work was founded by the French National Research Agency, the European Commission (Feder funds) and the Région Auvergne in the Framework of the LabEx IMobS^3.

$((P,T,\mathscr{A},w),\mathrm{cap})$ is able to reproduce the data when T contains enough transitions to perform for each $\mathbf{x}^j \in \mathscr{X}'$ the experimentally observed state change to $\mathbf{x}^{j+1} \in \mathscr{X}'$ and is called $\mathscr{X}'$-conformal. The goal is to determine all $\mathscr{X}'$-conformal capacitated Petri nets.

The approach of reconstructing Petri net models has been extended in two directions, for Petri nets with priorities on the transitions [12, 16, 19], and for extended Petri nets [3, 4] involving control-arcs.

The concept of priority relations among the transitions of a network was proposed in order to allow the modelization of deterministic systems. This means that for states where at least two transitions are enabled, the decision between the different alternatives is not taken randomly, but the transition with highest priority is forced to switch.

An extended Petri net is a Petri net with additionally two types of control-arcs; read- and inhibitor-arcs, which allow to disable transitions in certain system states. Hence, extended Petri nets use control-arcs as additional activation rules to prevent enabled transitions from switching.

An *$\mathscr{X}'$-deterministic extended Petri net* is a capacitated, extended Petri net with priorities on the transitions that is able to correctly reproduce a given set of experimental time-series data $\mathscr{X}'$. In particular, in every observed state $\mathbf{x}^j \in \mathscr{X}'$, exactly the transition has highest priority, which is required to obtain the observed successor state $\mathbf{x}^{j+1} \in \mathscr{X}'$.

Already in the case of reconstructing standard Petri nets [6], standard Petri nets with priorities [12] or extended Petri nets without priorities [3], there is typically no unique Petri net being able to reproduce the given data, but a large set of solution alternatives. We expect that reconstructing $\mathscr{X}'$-deterministic extended Petri nets results in even larger solution sets. To keep the solution set small, while still guaranteeing completeness, the idea is to generate only Petri nets being minimal in the sense that all other nets fitting the data contain the reconstructed ones.

While minimality can be easily related to set inclusion of the transition sets of standard Petri nets, the difficulty for extended Petri nets with priorities is that priority relations and control-arcs are concurrent concepts to prevent enabled transitions from switching.

Our contribution is to define a notion of minimality taking both concepts into account. For that, we define when two $\mathscr{X}'$-deterministic extended Petri nets are equivalent, and provide an inclusion relation for such networks. This allows us to give a definition for minimal $\mathscr{X}'$-deterministic extended Petri nets and to classify them by their set of transitions and their set of control-arcs (see Section 3).

2 Petri Nets

A *Petri net* $\mathscr{P} = (P,T,\mathscr{A},w)$ is a weighted directed bipartite graph with two kinds of nodes, places and transitions. The places $p \in P$ represent the system components (e.g. proteins, enzymes, genes, receptors or their conformational states) and the transitions $t \in T$ stand for their interactions (e.g., chemical reactions, activations or causal dependencies). The arcs in $\mathscr{A} \subset (P \times T) \cup (T \times P)$ link places and transitions, and the arc weights $w : \mathscr{A} \to \mathbb{N}$ reflect stoichiometric coefficients of the reactions.

A *state* in a Petri net $(P,T,\mathscr{A},w)$ is a vector $\mathbf{x} \in \mathbb{N}^{|P|}$, where each entry x_p corresponds to a place $p \in P$. In biological system, the components usually can be considered to be bounded. This leads to the definition of capacitated Petri nets. A *capacitated Petri net* $(\mathscr{P},\mathrm{cap})$ is a Petri net $\mathscr{P}$ together with a capacity function $\mathrm{cap} : P \to \mathbb{N}$, restricting the possible state space to $\mathscr{X} = \{\mathbf{x} \in \mathbb{N}^{|P|} \mid x_p \leq \mathrm{cap}(p)\}$.

Extended Petri nets $(P,T,A_S \cup A_R \cup A_I,w)$ have in addition to the set of standard-arcs A_S two further sets of arcs, the set of *read-arcs* A_R and the set of *inhibitor-arcs* A_I. The set $A_C = A_R \cup A_I$ is called the set of *control-arcs*.

A transition $t \in T$ is said to be *enabled* in a state $\mathbf{x} \in \mathscr{X}$ if all of the following conditions hold:

(i) $x_p \geq w(p,t)$ for all p with $(p,t) \in A_S$,

(ii) $x_p + w(t,p) \leq \mathrm{cap}(p)$ for all p with $(t,p) \in A_S$,

(iii) $x_p \geq w(p,t)$ for all p with $(p,t) \in A_R$,

(iv) $x_p < w(p,t)$ for all p with $(p,t) \in A_I$.

We say that a transition t is *disabled* in $\mathbf{x}$, if t is not enabled in $\mathbf{x}$. If a transition $t \in T$ is enabled in a state $\mathbf{x} \in \mathbb{N}^{|P|}$ it may *switch* or *fire*, leading to a successor state $\mathbf{x}' \in \mathbb{N}^{|P|}$, with

$$x'_p = \begin{cases} x_p - w(p,t), & \text{for all } p \text{ with } (p,t) \in A_S, \\ x_p + w(t,p), & \text{for all } p \text{ with } (t,p) \in A_S, \\ x_p, & \text{otherwise.} \end{cases}$$

In an extended Petri net there is usually more than only one transition enabled in a state. From all enabled transitions, one is chosen non-deterministically to fire in that state. However, this is not appropriate for biological or other deterministic systems, where a certain stimulation always results in the same response. For this purpose priorities on transitions can be used. We call $(\mathscr{P}, \mathscr{O})$ an *extended Petri net with priorities*, if $\mathscr{P}$ is an extended Petri net and $\mathscr{O}$ a priority relation on T. In this work, a *priority relation* is a partial order on the set of transitions. Another possibility for defining a priority relation is given in [16].

A capacitated, extended Petri net with priorities on transitions $(\mathscr{P}, \mathscr{O})$ which is able to reproduce a given experimental time-series data $\mathscr{X}'$ is called *$\mathscr{X}'$-deterministic extended Petri net*.

3 Classification of $\mathscr{X}'$-Deterministic Extended Petri Nets

In this section we consider $\mathscr{X}'$-deterministic extended Petri nets, generated by reconstruction from experimental time-series data $\mathscr{X}'$ (see [18] for more information). To keep the solution set as small as possible, while ensuring its completeness, we are interested in getting only those $\mathscr{X}'$-deterministic extended Petri nets, which are minimal in the sense that all other possible nets contain the generated ones. We propose to formalize an inclusion relation between $\mathscr{X}'$-deterministic extended Petri nets on classes of equivalent Petri nets.

In the literature there are several concepts for the equivalence of Petri nets, two often used ones are marking equivalence[1] (see, e.g., [7]) and bisimulation[2] equivalence (see, e.g., [10, 14]). These two concepts are not suitable for our purpose, as we need to compare two X'-deterministic extended Petri nets only taking states in X' into account, but starting from all initial states $\mathbf{x}_1 \in X$, not only for one initial state.

In general, the reconstruction approach generates Petri nets not only from the observed sequences, but also considers possible intermediate states between two observed states, to ensure the completeness of the solution set (see, e.g., [17]). This implies that for two $\mathscr{X}'$-deterministic Petri nets not only the set of control-arcs and priorities can be different, but also the set of places, transitions and standard-arcs.

[1] Let $\mathscr{P} = (P, T, \mathscr{A}, w)$ be a Petri net equipped with an initial state. A sequence of transitions $t^1, \ldots, t^k$ in T is called *feasible switching sequence* for a state $\mathbf{x}^0 \in \mathscr{X}$ in $\mathscr{P}$, if $\mathbf{x}^j \in \mathscr{X}$ is the successor state of $\mathbf{x}^{j-1}$ switching the transition t^j, for all $1 \leq j \leq k$. A state $\mathbf{x}^k \in \mathscr{X}$ is *reachable* from an initial state $\mathbf{x}^0 \in \mathscr{X}$ if there exists a feasible switching sequence from $\mathbf{x}^0$ to $\mathbf{x}^k$. Two Petri nets are *marking equivalent*, if they have the same set of reachable states, starting from the same initial state.

[2] Two Petri nets are *bisimilar* if every feasible switching sequence in one Petri net is a feasible switching sequence in the other Petri net, and vice versa.

Definition 1. Two $\mathscr{X}'$-deterministic extended Petri nets $(\mathscr{P},\mathrm{cap},\mathscr{O})$ and $(\hat{\mathscr{P}},\mathrm{cap},\hat{\mathscr{O}})$ are *$\mathscr{X}'$-equivalent* if $P=\hat{P}$, $T=\hat{T}$, $A_S=\hat{A}_S$ and $w_{|A_S}=\hat{w}_{|\hat{A}_S}$ hold.

One can easily verify that this indeed defines an equivalence relation. On the induced equivalence class we give an inclusion relation based on the concept of bisimulation equivalence. For that, we need to introduce a further notion. Let $(\mathscr{P},\mathrm{cap},\mathscr{O})$ be an $\mathscr{X}'$-deterministic extended Petri net; we call a sequence of transitions $t^1,\ldots,t^k$ *$\mathscr{O}$-feasible switching sequence* for a state $\mathbf{x}^0$ if $t^j\in\mathscr{T}_{\mathscr{P},\mathscr{O}}(\mathbf{x}^{j-1}):=\{t\in T\mid t \text{ enabled in } \mathbf{x}^{j-1}\wedge\nexists t'\in T \text{ with } t' \text{ enabled in } \mathbf{x}^{j-1} \text{ and } (t<t')\in\mathscr{O}, t\neq t'\}$, for all $1\le j\le k$.

Definition 2. Consider two $\mathscr{X}'$-equivalent Petri nets $(\mathscr{P},\mathrm{cap},\mathscr{O})$ and $(\hat{\mathscr{P}},\mathrm{cap},\hat{\mathscr{O}})$. We say $\mathscr{P}$ is *included* in $\hat{\mathscr{P}}$, denoted by $\mathscr{P}\subseteq\hat{\mathscr{P}}$, if and only if for all states $\mathbf{x}\in\mathscr{X}$, every $\hat{\mathscr{O}}$-feasible switching sequence for $\mathbf{x}$ in $\hat{\mathscr{P}}$ is a $\mathscr{O}$-feasible switching sequence for $\mathbf{x}$ in $\mathscr{P}$.

In Figure 1 two $\mathscr{X}'$-equivalent Petri nets $(\mathscr{P},\mathbb{1},\mathscr{O})$ and $(\hat{\mathscr{P}},\mathbb{1},\hat{\mathscr{O}})$ are shown, where $\mathscr{X}'$ is given by

$$(1\,1\,0\,0)\to(1\,0\,0\,1)\to(0\,0\,1\,1) \text{ and } (1\,0\,0\,0)\to(0\,0\,1\,0).$$

In this example we have $\mathscr{P}\subseteq\hat{\mathscr{P}}$.

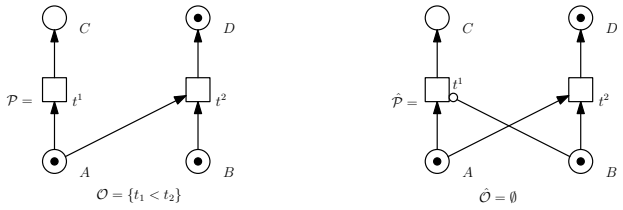


Figure 1: Consider two $\mathscr{X}'$-equivalent Petri nets, $(\mathscr{P},\mathbb{1},\mathscr{O})$ and $(\hat{\mathscr{P}},\mathbb{1},\hat{\mathscr{O}})$. For every state $\mathbf{x}\in\mathscr{X}\setminus\{(1\,1\,0\,1)\}$ both nets have the same $\mathscr{O}$-feasible switching sequence (resp. $\hat{\mathscr{O}}$-feasible switching sequence) for $\mathbf{x}$. However, in the state $\mathbf{x}'=(1\,1\,0\,1)$, which is indicated as marking in the figure, we have $\mathscr{T}_{\mathscr{P},\mathscr{O}}(\mathbf{x}')=\{t^1\}$ but $\mathscr{T}_{\hat{\mathscr{P}},\hat{\mathscr{O}}}(\mathbf{x}')=\emptyset$. Therefore, $\mathscr{P}\subseteq\hat{\mathscr{P}}$ follows.

In order to formulate a minimality concept, we need the following notions. For an $\mathscr{X}'$-deterministic extended Petri net $((P,T,A_S\cup A_C,w),\mathrm{cap},\mathscr{O})$,

- a control-arc $(p,t)\in A_C$ is *necessary* if $((P,T,A_S\cup(A_C\setminus\{(p,t)\}),w),\mathrm{cap},\mathscr{O})$ is not $\mathscr{X}'$-conformal;
- a priority $(t<t')\in\mathscr{O}$ is *necessary* if $\mathscr{O}\setminus\{(t<t')\}$ is not a partial order or $(\mathscr{P},\mathrm{cap},\mathscr{O}\setminus\{(t<t')\})$ is not $\mathscr{X}'$-conformal;
- a priority $(t<t')\in\mathscr{O}$ is *strictly necessary* if $(\mathscr{P},\mathrm{cap},\mathscr{O}\setminus\{(t<t')\})$ is not $\mathscr{X}'$-conformal.

If a priority is not (strictly) necessary, we call this element (strictly) *unnecessary*.

Definition 3. Among all $\mathscr{X}'$-equivalent extended Petri nets, $(\mathscr{P},\mathrm{cap},\mathscr{O})$ is *minimal* if and only if $(\mathscr{P},\mathrm{cap},\mathscr{O})$ does neither have unnecessary elements nor another $\mathscr{X}'$-deterministic extended Petri net $(\hat{\mathscr{P}},\mathrm{cap},\hat{\mathscr{O}})$ being $\mathscr{X}'$-equivalent to $(\mathscr{P},\mathrm{cap},\mathscr{O})$ is included in $\mathscr{P}$.

Based on this inclusion relation for $\mathscr{X}'$-equivalent Petri nets, we compare $\mathscr{X}'$-deterministic extended Petri nets by distinguishing the following four cases:

(i) $\mathscr{O}\subset\hat{\mathscr{O}}$ and $A_C=\hat{A}_C$,
(ii) $\mathscr{O}=\hat{\mathscr{O}}$ and $A_C\subset\hat{A}_C$,
(iii) $\mathscr{O}\subset\hat{\mathscr{O}}$ and $A_C\subset\hat{A}_C$,
(iv) $\hat{\mathscr{O}}\subset\mathscr{O}$ and $A_C\subset\hat{A}_C$.

The idea behind this classification is to determine, when we can safely remove a priority or a control-arc in order to get a "smaller" $\mathscr{X}'$-deterministic extended Petri net.

Theorem 4 (Case (i))**.** *Let $(\mathscr{P},\mathrm{cap},\mathscr{O})$ and $(\hat{\mathscr{P}},\mathrm{cap},\hat{\mathscr{O}})$ be two $\mathscr{X}'$-equivalent Petri nets with $\mathscr{O}\subset\hat{\mathscr{O}}$ and $A_C=\hat{A}_C$. Then $\mathscr{P}\subseteq\hat{\mathscr{P}}$ holds.*

One can find examples, so that in the cases (ii) and (iii) neither $\mathscr{P} \subseteq \hat{\mathscr{P}}$ nor $\hat{\mathscr{P}} \subseteq \mathscr{P}$ follows, i.e., both $\mathscr{X}'$-deterministic extended Petri nets cannot be compared w.r.t. minimality. In this case, possibly both Petri nets are minimal.

In Figure 2 one can see an example for two $\mathscr{X}'$-equivalent Petri nets, with $\mathscr{X}'$ given by the sequences

$$(1\,0\,0\,0\,0) \to (0\,0\,0\,1\,0),\ (0\,1\,1\,0\,0) \to (0\,1\,0\,0\,1) \text{ and } (1\,1\,1\,0\,0) \to (1\,1\,0\,0\,1) \to (0\,1\,0\,1\,1).$$

In this example, case (ii) holds, i.e., $\mathscr{O} = \hat{\mathscr{O}}$ and $A_C \subset \hat{A}_C$, but neither $\mathscr{P} \subseteq \hat{\mathscr{P}}$ nor $\hat{\mathscr{P}} \subseteq \mathscr{P}$ holds.

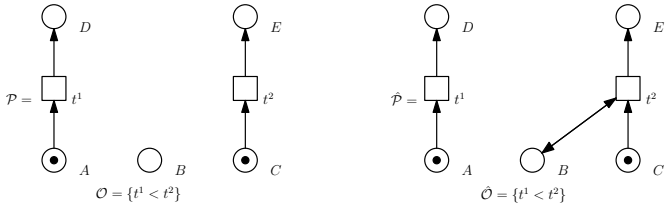


Figure 2: Consider two $\mathscr{X}'$-equivalent Petri nets, $(\mathscr{P}, \mathbb{1}, \mathscr{O})$ on the left and $(\hat{\mathscr{P}}, \mathbb{1}, \hat{\mathscr{O}})$ on the right. Both nets have the same set of priorities on their transitions, but $A_C \subset \hat{A}_C$. In the state $\mathbf{x} = (1\,0\,1\,0\,0)$, shown in the figure, we see that in $\mathscr{P}$ both transitions are enabled, but due to the priority $(t^1 < t^2)$ only t^2 is allowed to switch. In $\hat{\mathscr{P}}$ the transition t^2 is disabled. Therefore t^1 switches in $\mathbf{x}$. It follows that neither the switching sequences starting from $\mathbf{x}$ in $\mathscr{P}$ is included in the switching sequence starting from $\mathbf{x}$ in $\hat{\mathscr{P}}$, nor vice versa; thus, neither $\mathscr{P} \subseteq \hat{\mathscr{P}}$ nor $\hat{\mathscr{P}} \subseteq \mathscr{P}$ holds.

Conjecture 1 (Case (iv))**.** Let $(\mathscr{P}, \mathrm{cap}, \mathscr{O})$ and $(\hat{\mathscr{P}}, \mathrm{cap}, \hat{\mathscr{O}})$ be two $\mathscr{X}'$-equivalent Petri nets with $\hat{\mathscr{O}} \subset \mathscr{O}$ and $A_C \subset \hat{A}_C$. Furthermore, let every control-arc in $\hat{A}_C$ be necessary.

If for all $(t < t') \in \mathscr{O} \setminus \hat{\mathscr{O}}$ the following properties hold:

- there exists an inhibitor-arc $(p,t) \in \hat{A}_I \setminus A_I$, where p is a pre-place of t' in $\mathscr{P}$, or there exists a read-arc $(p',t) \in \hat{A}_R \setminus A_R$, where p' is a post-place of t' in $\mathscr{P}$,
- $(t < t')$ is strictly necessary in $\mathscr{O}$,
- there does not exist a transition t'' with $(t'' < t) \in \hat{\mathscr{O}}$,
- t and t' are not both enabled in any state in $\hat{\mathscr{P}}$,

then $\mathscr{P} \subseteq \hat{\mathscr{P}}$ follows.

4 Conclusion

In this work, we address the problem of classifying capacitated extended Petri nets with priorities, reconstructed from experimental time-series data $\mathscr{X}'$.

We defined both, an equivalence relation for such Petri nets, as well as an inclusion relation on the networks within an equivalence class in order to identify minimal elements therein. For that, we distinguished four cases for inclusions of $\mathscr{O}, \hat{\mathscr{O}}$ and $A_C, \hat{A}_C$.

Obviously, the condition for Theorem 4 can be checked easily. The first condition of Conjecture 1 can be quickly tested, while the second condition is ensured by the reconstruction algorithm itself. The other two conditions can be done in polynomial time by [16]. Hence, these conditions imply an inclusion of two $\mathscr{X}'$-equivalent Petri nets that could indeed be applied practically to reduce the solution set of the studied reconstruction approach.

Our further goals are to prove Conjecture 1 and to identify some properties for capacitated extended Petri nets with priorities, so that also for the two cases (ii) and (iii) some sufficient conditions for their inclusion can be imposed.

References

[1] M. Chen and W. Hofestädt. A Petri net application of metabolic processes. *J. Syst. Anal. Modell. Simul.*, 16:113–122, 1994.

[2] M. Chen and W. Hofestädt. Quantitative Petri net model fo gene regulated metabolic networks in the cell. *In Silico Biology*, 3:347–365, 2003.

[3] M. Durzinsky, W. Marwan, and A. K. Wagler. Reconstruction of extended Petri nets from time series data and its application to signal transduction and to gene regulatory networks. *BMC Systems Biology*, 5, 2011.

[4] M. Durzinsky, W. Marwan, and A. K. Wagler. Reconstruction of extended Petri nets from time-series data by using logical control functions. *Journal of Mathematical Biology*, 2012. `http://dx.doi.org/10.1007/s00285-012-0511-3`.

[5] M. Durzinsky, A. K. Wagler, and R. Weismantel. A combinatorial approach to reconstruct Petri nets from experimental data. In Monika Heiner and Adelinde M. Uhrmacher, editors, *CMSB*, volume 5307 of *Lecture Notes in Computer Science*, pages 328–346. Springer, 2008.

[6] M. Durzinsky, A. K. Wagler, and R. Weismantel. An algorithmic framework for network reconstruction. *Journal of Theoretical Computer Science*, 412(26):2800–2815, 2011.

[7] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets - a Survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[8] B. N. Kholodenko, A. Kiyatkin, F. J. Bruggeman, E. Sontag, H. V. Westerhoff, and J. B. Hoek. Untangling the wires: A strategy to trace functional interactions in signaling and gene networks. *Proceedings of the National Academy of Sciences*, 99(20):12841–12846, 2002.

[9] I. Koch and M. Heiner. Petri nets. In B. H. Junker and F. Schreiber, editors, *Biological Network Analysis*, Wiley Book Series on Bioinformatics, pages 139–179, 2007.

[10] M. Kot and Z. Sawa. Bisimulation equivalence of a bpp and a finite-state system can be decided in polynomial time. *Electron. Notes Theor. Comput. Sci.*, 138(3):49–60, 2005.

[11] R. Laubenbacher and B. Stigler. A computational algebra approach to reverse engineering of gene regulatory networks. *Journal of Theoretical Biology*, 229:523–537, 2005.

[12] W. Marwan, A. K. Wagler, and R. Weismantel. A mathematical approach to solve the network reconstruction problem. *Math. Methods of Operations Research*, 67(1):117–132, 2008.

[13] W. Marwan, A. K. Wagler, and R. Weismantel. Petri nets as a framework for the reconstruction and analysis of signal transduction pathways and regulatory networks. *Natural Computing*, 10:639–654, 2011.

[14] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[15] J. W. Pinney, R. D. Westhead, and G. A. McConkey. Petri net representations in systems biology. *Biochem. Soc. Tarns.*, 31:1513–1515, 2003.

[16] L. M. Torres and A. K. Wagler. Encoding the dynamics of deterministic systems. *Math. Methods of Operations Research*, 73:281–300, 2011.

[17] A. K. Wagler. *Prediction of network structure*, volume 16 of *Computational Biology*, pages 309–338. Springer London, 2010.

[18] A. K. Wagler and J.-T. Wegener. On minimality and equivalence of Petri nets. *Proceedings of Concurrency, Specification and Programming CS&P'2012 Workshop*, 2:382–393, 2012.

[19] A. K. Wagler and R. Weismantel. The combinatorics of modeling and analyzing biological systems. *Natural Computing*, 10:655–681, 2011.